Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Håvard Hunshamar
Marcus Frenje

Techniques and tools for cryptocurrency intelligence and blockchain forensics

Master's thesis in Communication Technology and Digital Security
Supervisor: Danilo Gligoroski
June 2022

Norwegian University of
Science and Technology

Håvard Hunshamar
Marcus Frenje

Techniques and tools for cryptocurrency intelligence and blockchain forensics

Master’s thesis in Communication Technology and Digital Security
Supervisor: Danilo Gligoroski
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Title:	Techniques and tools for cryptocurrency intelligence and blockchain forensics
Students:	Håvard Hunshamar & Marcus Frenje

Problem description:

Bitcoin was released in 2009 as the first decentralized cryptocurrency [1], introducing the concepts of decentralization and anonymity to online payments through blockchain technology. However, the degree of anonymity is not as prevalent as it may seem. A key part of decentralization is the use of ledgers, where all transactions are listed as public information [2]. Bitcoin addresses are used as identifiers in the transactions, which are not directly linked to real-life identities but can be analyzed and deanonymized to varying degrees. Several other decentralized cryptocurrencies have been developed, some with advanced privacy-enhancing technologies that obfuscate the transaction details and make deanonymization more difficult [3]. Cryptocurrencies are increasingly used in illicit transactions [4], which has created a huge market for analytics tools used by law enforcement agencies for investigation purposes. Users can be linked directly to transactions based on public information on the blockchain using address clustering techniques, graph learning technologies, and network analysis. In recent years, several tools have been developed; one of them is GraphSense, which will be the focus of this thesis. This tool is studied and scrutinized to identify possible improvements in its deanonymization algorithms. These algorithms are assessed based on the reductions that can be made to the number of possible identities involved in a set of transactions.

Date approved:	2022-02-21
Responsible professor:	Danilo Gligoroski, IIK
Supervisor(s):	Danilo Gligoroski, IIK

Abstract

Address clustering breaks the pseudonymity of cryptocurrencies by linking multiple addresses controlled by one user to one entity. In this thesis, we analyze existing clustering heuristics based on *common-input-ownership* and one-time change addresses for Bitcoin transactions. We propose new heuristics based on an optimal combination of properties and implement them with Graphsense Cryptoanalytics platform, Python, and MongoDB on two data sets with real transactions from the Bitcoin blockchain. Our proposed heuristics aim to achieve adequate reduction ratios, while evading false positive results by using strict properties for defining OTC addresses and not including transactions with *coin mixing* characteristics in the clustering.

Graphsense Cryptoanalytics platform's built-in clustering method is the standard *common-input-ownership heuristic* which includes all transactions when performing address clustering. Our redefined *common-input-ownership heuristic* combined with our fairly strict OTC address heuristic achieves 71.30% of the address reduction ratio, but with a definite lower occurrence of false positive results. The experimental results make a strong case for Graphsense to change its address clustering method and switch to our variation, which provides more reliable results.

We also looked into methods for clustering transactions obfuscated by *coin mixing* strategies, mainly for shared CoinJoin transactions. Our proposed clustering heuristics can be applied to sub-transactions detected by linking the sums of input and output values in CoinJoin transactions. A method for achieving this is discussed, but no experimentation was conducted due to the complexity of the problem.

Sammendrag

Ved å gruppere flere kryptovaluta-adresser til en bruker basert på informasjon som er tilgjengelig på blokkjeden er anonymiteten for transaksjoner svært sårbar. I denne masteroppgaven undersøker vi eksisterende heuristikker for adressegruppering basert på felles eierskap av input-adresser og på å finne OTC-adresser – adresser generert for å motta vekslepenger i transaksjoner. Vi foreslår nye heuristikker basert på en optimal kombinasjon av eksisterende heuristikker og andre egenskaper, og implementer dem med Cryptoasset Analytics Platform, Python, og MongoDB. Vi bruker to datasett med ekte transaksjoner fra Bitcoin sin blokkjede for å teste heuristikkene. Våre foreslåtte heuristikker er laget med mål om å oppnå tilstrekkelige reduksjonsforhold og unngå falske positive resultater. Dette blir oppnådd ved å bruke strenge krav for hva som definerer en OTC-adresse, og ved å ikke inkludere transaksjoner med egenskaper tilhørende *coin mixing* transaksjoner.

Cryptoasset Analytics Platform har en innebygget grupperingsmetode. Denne er en standard heuristikk for felles eierskap av input adresser, som inkluderer alle transaksjoner når adressegruppering blir utført. Vår redefinerte felles input eierskap heuristikken kombinert med vår strenge OTC-addresse heuristikk oppnår 71.30% av adressereduksjonsforholdet, men med en utvilsomt lavere forekomst av falsk-positive resultater. De eksperimentelle resultatene argumenterer for at Graphsense burde endre sin adressegrupperingsmetode til vår foreslåtte variant, som gir mer pålitelige resultater.

Vi har også sett på metoder for å gruppere transaksjoner obfuskert av *coin mixing*-strategier, hovedsakelig CoinJoin-transaksjoner. Vår foreslåtte grupperingsheuristikk kan brukes på deltransaksjoner avdekket av å studere summene som overføres i input- og outputverdiene. En metode for å utføre dette er diskutert, men ikke eksperimentert med på grunn av problemets kompleksitet.

Preface

This is a master thesis submitted by the Norwegian University of Science and Technology (NTNU). It is the final work of our 5-year study program in Communication Technology and Digital Security, conducted from January to June 2022.

We want to thank our supervisor and responsible professor Danilo Gligoroski for his tremendous help along the way. He has been very enthusiastic about our project from start to finish and has believed in us until the end.

We also want to thank our friends and family, especially Johan Frenje, for taking their time to read our thesis and provide excellent feedback.

Finally, we want to thank the Graphsense team for letting us use their demo API and for increasing our hourly request limit from 1,000 to 10,000 during the final weeks of research.

Contents

List of Figures xi

List of Tables xiii

List of Algorithms xvi

1 Introduction 1
1.1 About Bitcoin . . . 1
1.2 Motivation . . . 2
1.3 Methodology . . . 3
1.4 Related work . . . 4
1.4.1 Transaction graph analysis . . . 4
1.4.2 Deanonymization based on network analysis . . . 5
1.4.3 Detecting fraudulent transactions with the K-means algorithm 5
1.5 Scope of the thesis . . . 6
1.6 Ethical concerns . . . 7

2 Tools and technology 9
2.1 GraphSense Cryptoasset Analytics Platform . . . 9
2.2 Virtual Machine . . . 10
2.3 MongoDB . . . 10
2.3.1 Robo 3T . . . 10
2.4 Python . . . 11
2.4.1 Packages . . . 11
2.4.2 Version control - git . . . 12
2.5 Data sets . . . 12

3 Theory background 15
3.1 The structure of Bitcoin transactions . . . 15
3.1.1 Bitcoin Client . . . 15
3.1.2 Bitcoin Wallet . . . 15
3.1.3 Outputs . . . 15
3.1.4 Inputs . . . 15

3.1.5 Transaction fees 16
3.2 Bitcoin Address clustering 17
3.2.1 Common-input-ownership 17
3.2.2 One-time Change Address 19
3.2.3 Combination of common-input-ownership heuristic and OTC heuristic 26
3.3 Difficulties in performing address clustering 26
3.3.1 Unique key pairs 26
3.3.2 Coin mixing 27
3.3.3 CoinJoin 27

4 Implementation details 31
4.1 Initial setup 31
4.1.1 Python 31
4.1.2 Graphsense 32
4.1.3 Virtual Machine 33
4.2 Data sets 33
4.2.1 Data set 1 35
4.2.2 Data set 2 35
4.3 Common-input-ownership heuristics 36
4.4 One-time change address heuristics 38
4.4.1 Coin generation and the number of inputs and outputs 38
4.4.2 Occurrence of self-change address 39
4.4.3 Address reused as output 39
4.4.4 Number of digits in OTC value 40
4.4.5 Optimal change heuristic 41
4.4.6 Determining the OTC output 41
4.5 Combination of common-input-ownership heuristic and OTC heuristic 42
4.6 Testing 43

5 Results 45
5.1 Common-input-ownership heuristics 46
5.2 One-time change heuristics 47
5.3 Combination of common-input-ownership heuristic and OTC heuristic 48
5.3.1 Reduction over time 49

6 Future work 53
6.1 Difficulties in detecting centralized coin mixing 53
6.2 Analyzing a larger quantity of transactions 54
6.3 Combining clustering with direct deanonymization techniques 55
6.4 The CoinJoin sub-transaction problem 55

7 Conclusion **59**

References **61**

Appendices

A Communication with the Graphsense team **67**

B Request for resources in Openstack **70**

C Implementations **71**

C.1 Populating data sets 71
C.2 Common-input-ownership heuristics 75
C.3 One-time change address heuristics 78
C.4 Combination of the common-input-ownership heuristic and OTC heuristic 86
C.5 Testing 87

List of Figures

1.1 Volume of cryptocurrencies used in illicit transactions [4] 3

2.1 Visualization of a MongoDB document based on a query - a Bitcoin transaction with an equal number of inputs and outputs 11
2.2 Transactions in data set 1. Total number of transactions is 73,798 . . . 12

3.1 Bitcoin transactions including fees 16
3.2 Common-input-ownership 17
3.3 Occurrences of transactions with multiple inputs and one output. 18
3.4 The change output 19
3.5 Number of inputs for transaction in data set 1 24
3.6 Combination of common-input-ownership and OTC 26
3.7 Illustration of a centralized coin mixing scheme 28
3.8 Illustration of the concept behind CoinJoin 28
3.9 Example of sub-transactions within a CoinJoin transaction 29

5.1 Reduction ratio of the common-input-ownership heuristics on data set 1 46
5.2 Impact of OTC properties on data set 1 47
5.3 Volume of transactions with OTC address found using different heuristics on data set 1. Total number of transactions: 63,783 48
5.4 Performance of address clustering heuristics on data set 1 49
5.5 Performance of address clustering heuristics on a growing volume of transactions on data set 2 50

6.1 Example of a mixing transaction through a centralized mixing server . . 54
6.2 Example of a perfect CoinJoin transaction 57

List of Tables

5.1 Results of address clustering heuristics on data set 1 49
5.2 Results of address clustering heuristics on a growing volume of transactions on data set 2 51

List of Algorithms

3.1 Find all possible subset sums in a list of values 30

4.1 Accessing the API key's environmental variable 31

4.2 Graphsense python client example - transactions for block with height 500,000 32

4.3 Pymongo example - query for transactions with multiple inputs and insert into a new collection 33

4.4 Get all transactions from a block with inputs and outputs 34

4.5 Given a block height, find surrounding blocks to reach desired number of transactions 35

4.6 Find the Bitcoin block closest to a given timestamp 36

4.7 Create entities with shared input addresses from a list of transactions 37

4.8 Create entities for each input address from a list of transactions . . . 38

4.9 Coin generation, number of inputs and number of outputs 39

4.10 Check for self-change address in the transaction 39

4.11 Check if an output is reused as an output in a later transaction . . . 40

4.12 Check if the decimal representation of the OTC value has more than four digits after the dot 41

4.13 Optimal change heuristic 41

4.14 Determining the OTC output 42

4.15 Reduce the clusters of the entities made with the common-input-ownership heuristic using OTC addresses found with heuristic 2.4 . . 43

4.16 Unit testing on Algorithm 4.10 44

Chapter 1
Introduction

1.1 About Bitcoin

Bitcoin is the first-ever decentralized cryptocurrency, introduced by Satoshi Nakamoto in 2008 [1]. It is currently the most used cryptocurrency in the world [5]. Bitcoin was the first payment system to use a fully decentralized architecture, where distributed systems (or miners) are used to confirm payments.

Decentralization is achieved through the use of public ledgers in the form of blockchains [5]. The blockchain is distributed across a network of nodes running the cryptocurrency's client to ensure a secure and decentralized record of transactions [6]. Transactions are confirmed by so-called miners - nodes that collaborate in adding blocks to the blockchain through proof-of-work [7]. This is achieved by forming a mining pool and combining the hardware effort of each miner to solve computationally hard problems. A mining reward in the form of currency is awarded and split among the participants of the mining pool to give an incentive to mine [5]. A cryptocurrency network will usually consist of several mining pools which compete against each other in solving problems [5].

A Bitcoin transaction always contains a number of input and output addresses, equivalent to the transaction's sender(s) and receiver(s). A user can control multiple addresses [8]. Each address stores a given Bitcoin value, and each input address will correspond to an output address from a previous transaction [9] (see section 3.1 for more details). The addresses do not correspond directly with a user identity; they are pseudonyms generated by a user's private key, otherwise known as Bitcoin addresses. However, it is considered impossible to retrieve a user's private key from a public Bitcoin address [10].

An address does not correspond directly with a user's identity. However, because all the transactions are available on public ledgers, Bitcoin's is pseudonymous [2]. An input address in a Bitcoin transaction always corresponds to the address of an output

from a previous transaction. This, along with the availability of transactions on public ledgers, make it easy to figure out transactional interactions between public Bitcoin addresses. This possibility has raised many questions regarding Bitcoin's anonymity. Later in this thesis, we show that it is possible to perform deanonymization techniques based on address clustering. This allows us to link multiple Bitcoin addresses to one user by inspecting transaction history and relating addresses to each other.

1.2 Motivation

The trustworthiness of the Bitcoin protocol has been questioned for a long time due to its pseudo-anonymity, and the potential to perform deanonymization attacks based on publicly available information on the blockchain [11]. Furthermore, the popularity of Bitcoin has exploded in the last decade, reflected in both the number of monthly transactions [12] and the significantly higher value as a currency since its release. However, the price of Bitcoin remains very volatile. Since starting this master thesis in January 2022, the price has decreased roughly 50% (as of June 2022).

The anonymity of Bitcoin has been an issue since its release and was even questioned in Satoshi Nakamoto's original Bitcoin paper, stating that further work on Bitcoin's anonymity was essential [1]. A few deanonymization attacks have also been proven possible. The most prominent ones involve relating addresses to one common entity, detecting anomalies, and eavesdropping on network traffic [13, 7, 14].

In response to the anonymity problem of Bitcoin, several privacy-focused cryptocurrencies have emerged, most notably Zcash and Monero. Zcash is an implementation of Bitcoin and was forked off the Bitcoin codebase in 2015 [3]. Zcash uses zero-knowledge proofs, a method for verifying information without revealing the actual information itself [15]. This way, transactions can be verified by anyone without revealing any transactional data [3]. Monero is another privacy-focused cryptocurrency that solves the traceability issues of Bitcoin by obfuscating addresses and amounts using ring signatures [3]. Nevertheless, attacks on Zcash and Monero have been proven possible through network analysis (see section 1.4.2).

Several anonymity-enhancing properties have also been proposed for Bitcoin, mainly *coin mixing* services. These services operate externally from Bitcoin and work without requiring changes to the Bitcoin core protocol [16]. *Coin mixers* obfuscate addresses and amounts within transactions (see section 3.3.2 for more details).

With more anonymity properties being implemented in cryptocurrencies, there is a bigger need than ever to investigate their potential socio-economic effects. The most dire consequence of increased anonymity is the use of cryptocurrencies in criminal activities such as money laundering, ransomware/scams/frauds, drug

trafficking, or even financing terrorism [4]. As shown in Figure 1.1, the volume of cryptocurrency used for illicit transactions is quite significant. Therefore, it is important to understand the potential market value of various tools to help detect such crimes. Deanonymization tools for public blockchains help law enforcement agencies to investigate illegal financial activities involving cryptocurrencies. For example, in 2020, the International Revenue Service (IRS) published a bounty of $625.000 for producing relevant results for investigations of the privacy-focused cryptocurrencies Monero and Lightning [17]. This bounty was awarded to the blockchain analytics firms Chainanalysis and Integra FEC, for tracing illegal activities within these cryptocurrencies [18].

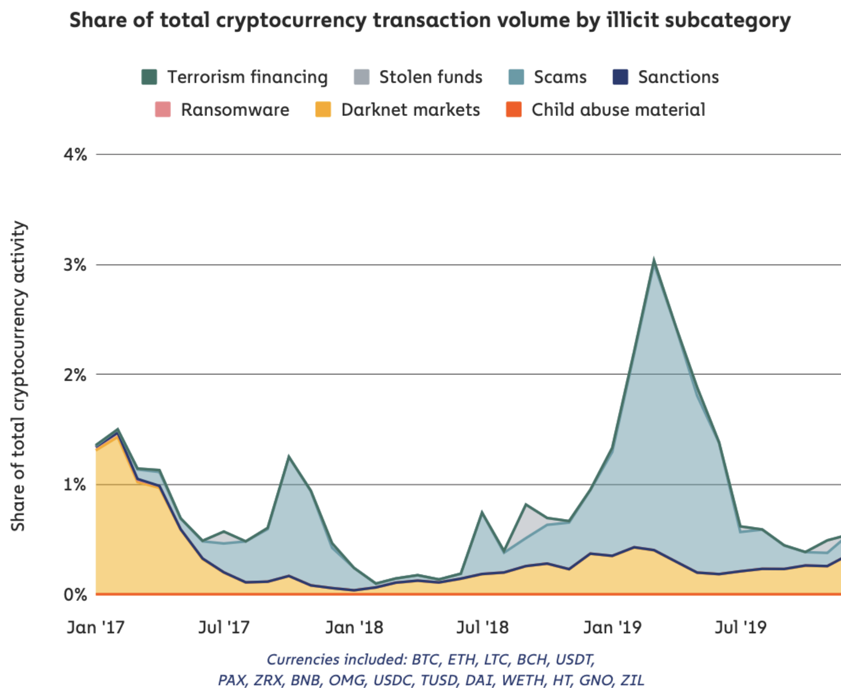


Figure 1.1: Volume of cryptocurrencies used in illicit transactions [4]

1.3 Methodology

Our work in this thesis was conducted using GraphSense Cryptoasset Analytics Platform. This open-source tool provides an interface for running advanced analytics on several cryptocurrencies. Graphsense performs address clustering on transactions using the standard *common-input-ownership heuristic* (see section 3.2.1). By using the built-in methods of Graphsense's semi-public demo API, we tested and analyzed existing clustering methods to find possible improvements in our proposed address

clustering heuristics. However, we also realized that *coin mixing* could pose a massive problem for address clustering.

In many cases, *coin mixing* strategies influence the clustering results that can be achieved. Therefore, we opted to try and exclude *coin mixed* transactions from our clustering (See chapter 3 for more details). This is feasible because most transactions do not use *coin mixers* [16]. However, the difficulty of performing deanonymization techniques on *coin mixing* transactions was considered and studied, as discussed in section 3.3.3.

1.4 Related work

In this thesis, we have explored a few specific deanonymization methods in Bitcoin in the form of address clustering. However, this is only one of many ways to perform deanonymization in Bitcoin and other cryptocurrencies. Since the Bitcoin blockchain is public, it is possible to create transaction graphs based on the transaction history available. This is known as graph analysis [19]. It is also possible to engage in the Bitcoin network and analyze IP traffic that the Bitcoin nodes broadcast [7]. Anomaly detection techniques on a dataset of Bitcoin transactions is also possible [14]. We briefly touch upon these strategies in this section.

1.4.1 Transaction graph analysis

A *transaction graph* illustrates the flow of cryptocurrency value between all Bitcoin addresses on a blockchain. It is based entirely on the history of available transactions. It can be used to analyze the transactional relationship between a selection of participants over time [20]. By constructing graphs of the transaction history, graph learning along with address reduction heuristics can be used to analyze them [19]. With a transaction graph, it is possible to study certain aspects of vertexes, such as degree, currency amount accumulated on the address, holding time of currency, series of active usage timestamps, and influx and efflux of the currencies of an address [19]. Furthermore, learning about the participants' behavior makes it possible to identify addresses belonging to large organizations. Such addresses, in most cases, have a very high vertex degree [19].

Reid and Harrigan [20] analyzed an alleged Bitcoin theft of 25,000 BTC, which took place in 2011. At the time, 25,000 BTC was worth roughly half a million dollars. With the help of transaction graphs and shortest path analysis between vertices, they could trace the flow of the stolen Bitcoin. They eventually figured out that the flow initially split between addresses would later merge, validating the likelihood of these addresses belonging to a single user [20].

1.4.2 Deanonymization based on network analysis

Biryukov, Khovratovich, and Pustogarov [7] show that it is possible to set up a Bitcoin core node and participate in the Bitcoin network as a passive eavesdropper. This is a feasible method for deanonymization for identifying the public IP addresses behind the participants by their established connection to the Bitcoin network. However, the attacker must maintain a connection to all participants in a network, as long as they are eavesdropping. Reid and Harrigan [20] also claimed that analyzing several public keys used simultaneously might reveal common ownership. This was later proven possible by Fanti and Viswanath [21], who showed a weakness in the transaction broadcasting protocol performed by participants in the Bitcoin network. By conducting a timestamp analysis on the network traffic broadcasted from a transaction, IP addresses can be linked to the pseudonyms used on the public blockchain. Biryukov and Tikhomirov [3] also proved that such timing attacks are possible on privacy-focused cryptocurrencies such as Zcash and Monero. Many Bitcoin users have resorted to using the anonymizing service Tor as a preventive measure against such attacks. However, Biryukov and Pustogarov [22] proved that this service enables new attacks that even a low-resource attacker can exploit. Attackers can act like a Man in the Middle and gain complete control of the information flows of all users who choose to use Bitcoin over Tor. This means both being able to learn their IP addresses, as well as stopping or delaying transactions and blocks [22].

1.4.3 Detecting fraudulent transactions with the K-means algorithm

The K-means algorithm is a heuristic method used to solve part of the NP-hard problem of clustering a number of data points into K clusters. It makes use of Euclidean distances, which is a mathematical measure for the difference between data points. In a cluster, this distance is to be minimized [14].

It is also possible to execute this algorithm on Bitcoin transactions by taking into account the following [14]:

1. Mean node degree of users transacted with
2. Variance of node degree of users transacted with
3. Mean transaction amount
4. Variance of transaction amount

Hirshman, Huang, and Macke [14] were able to run an implementation of the K-means algorithm on real Bitcoin transactions and detect users that behaved differently from the rest. These users performed transactions that did suggest some sort of money laundering scheme. However, there was no way to know for sure. A weakness with the K-means algorithm is that it usually requires a lot of data pre-processing and a-priori knowledge of the type of data to be analyzed. Having already defined data that points toward money laundering cases helps label future hypothetic money laundering cases [14]. In a later study by Monamo, Marivate, and Twala, [23], it is proven that higher accuracy detection of fraudulent transactions is possible by using labeled data from previously known cases. They also use a trimmed version of the K-means algorithm, which removes the most extreme values from the dataset and creates more representative clusters. A-priori knowledge of the optimal number of K clusters is also advantageous. Figuring this out is tedious and requires thorough studies and familiarity with the dataset. However, in 2020 a method of finding the optimal K value on a dataset was proposed by Sinaga, and Yang [24]. This method involves performing the algorithm in several iterations by initially creating a few but large clusters. In the later iterations, clustering is done within the clusters created during the earlier iterations [24].

1.5 Scope of the thesis

The scope of deanonymization of cryptocurrencies is enormous. We do not have the means to cover all aspects of this in this thesis. We have decided to focus only on Bitcoin because it is the most popular cryptocurrency worldwide. We do not cover any aspects related to other cryptocurrencies in the main part of this thesis. We only cover particular deanonymization techniques related to addressing clustering and reducing the number of possible entities. We do not cover any deanonymization methods mentioned in section 1.4. We also cover some anonymity enhancements related to Bitcoin, mainly *coin mixing* strategies, as well as methods of working around them.

1.6 Ethical concerns

There is a much sought-after need to investigate illicit transactions within cryptocurrencies. However, since maintaining a high level of privacy is crucial for the stability and longevity of cryptocurrencies, one must consider the balance between maintaining users' privacy and hindering illegal activities, as both cannot be fulfilled simultaneously. In this thesis, we propose a few alternate ways to perform deanonymization on Bitcoin transactions and share new ideas for working against anonymity-enhancing techniques such as *coin mixing*. We discourage malicious use of our proposed ideas, as it can potentially violate the privacy of Bitcoin users. The transactions analyzed in this thesis are real transactions, but we do not attempt to reveal any real identities. We also do not provide any information on specific transactions, only which blocks they belong to.

Chapter 2

Tools and technology

2.1 GraphSense Cryptoasset Analytics Platform

Graphsense [25] is a digital tool used for investigations of monetary flows and for running advanced customized analytics on cryptocurrency data gathered from public blockchains [26].

Graphsense was originally part of TITANIUM, a project run from May 1, 2017, to April 30, 2020, by a team of research organizations, industry partners, and law enforcement agencies [27]. The EU Commission funded TITANIUM with a total budget of 5 million euros to provide law enforcement agencies with tools capable of creating court-proof evidence of illicit activity involving cryptocurrencies [28]. When the project ended, six software tools had been developed, Graphsense being one of them. The Australian Institute of Technology (AIT) further developed Graphsense with funding from the Austrian Research Promotion Agency's KIRAS program KRYPTOMONITOR [29].

Graphsense provides a graph-centric perspective on cryptocurrency transactions, indicating the flow of currency between addresses over time [12]. While Graphsense supports multiple major cryptocurrencies such as Bitcoin, Bitcoin Cash, Litecoin, Zcash, and Ethereum [30], this thesis only focuses on Bitcoin. Through a REST-API, Graphsense offers data on all transactions ever performed and address sets - collections of Bitcoin addresses that are likely controlled by the same real-world entity [26]. The address sets are computed using address clustering methods and stored in the Graphsense database as entities. Graphsense currently applies the *common-input-ownership heuristic* exclusively for address clustering [26]. This heuristic assumes that input addresses used in the same transactions are controlled by one user [31]. This heuristic, along with its weaknesses and limitations, is explored in detail later in this thesis.

By contacting the Graphsense team [32], we got access to the semi-public Graph-

Sense demo through an API key. The number of requests was initially restricted to 1,000 per hour due to limited server capacity. However, the number was temporarily increased to 10,000 per hour during the final weeks of research. The communication with the Graphsense team is displayed in Appendix A.

The challenges of Graphsense remain in the growing volume, velocity, and semantically poor nature of the digital currency transaction data [12]. For example, Bitcoin alone processes around 250,000 transactions every day [33], causing scalability issues for crypto-analytics platforms such as Graphsense.

2.2 Virtual Machine

Through the Openstack (SkyHiGh) platform at NTNU [34], a Virtual Machine (VM) with 128GB RAM and 16 processor cores was assigned to this project. The request submitted to NTNU for Openstack resources is shown in Appendix B. The VM was accessed by linking it to the public part of an SSH key pair generated on our personal machines. Initially, the intention was to run the Graphsense projects on the VM. However, it was unnecessary as the Graphsense demo API, hosted on even more powerful hardware components [25], became available. So instead, the VM was used for running clustering algorithms and storing transactions and other relevant data on a MongoDB database.

2.3 MongoDB

MongoDB is a NoSQL database program that stores data in BSON format [35]. BSON stands for Binary JSON and is a binary serialization of JSON-like documents, allowing faster parsing, which results in faster querying and storage of data [36]. The MongoDB program was hosted on the VM, with one database containing multiple collections. Each collection contains one document for each entry, where an '_id' field is used as a unique primary key for each document.

2.3.1 Robo 3T

Robo 3T is a lightweight, open-source Graphical User Interface (GUI) for MongoDB, used to visualize and interact with collections and documents [37]. Robo 3T was used in this project for simple operations such as renaming and deleting collections, executing simple queries, and inspecting the relevant documents, as seen in Figure 2.1.

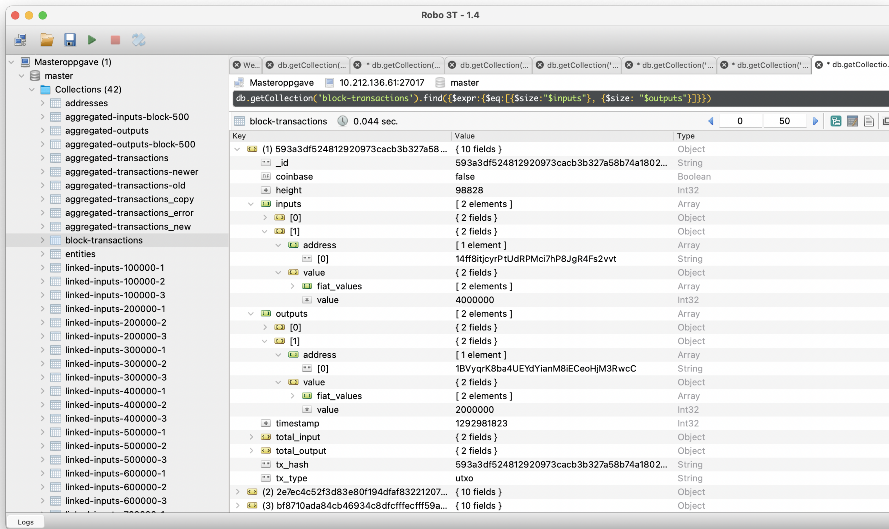


Figure 2.1: Visualization of a MongoDB document based on a query - a Bitcoin transaction with an equal number of inputs and outputs

2.4 Python

The Python programming language was chosen for this project. Python 3.9 [38] was used, with Visual Studio Code [39] as Integrated Development Environment (IDE). Python was chosen because of its high readability and ability to perform complex tasks with minimal code; it also has an extensive selection of packages and libraries which are applicable to this project.

2.4.1 Packages

Packages were installed using Pip [40], the standard package manager for Python, in a virtual environment. The following packages are the most relevant for this project.

Graphsense Python Client [41]: A Python interface for interacting with the Graphsense REST API and performing cryptocurrency analytics.

matplotlib [42]: A library for creating visualizations in Python, used for creating plots and graphs in this thesis.

pymongo [43]: A tool for working with mongoDB in Python.

dotenv [44]: A module that allows environment variables to be specified in a ".env" (dot-env) file.

2.4.2 Version control - git

Git [45] was used for collaboration and version control. It was also beneficial for accessing scripts from the VM. Github was used for hosting the git repository.

2.5 Data sets

For clustering and analytics, two datasets were used. First, a collection of transactions from various parts of Bitcoin's history was assembled. Then, a script was created to retrieve these transactions and relevant data from Graphsense and store it in MongoDB for easy accessibility. The Bitcoin blockchain [46] consists of over 700,000 immutable chronologically ordered blocks, each containing a varying number of transactions [47]. For each 100,000th block, the surrounding blocks containing 10,000 transactions were included in the data set.

This resulted in *data set 1*: 73,798 transactions, ∼10,000 from each ∼100,000 block:

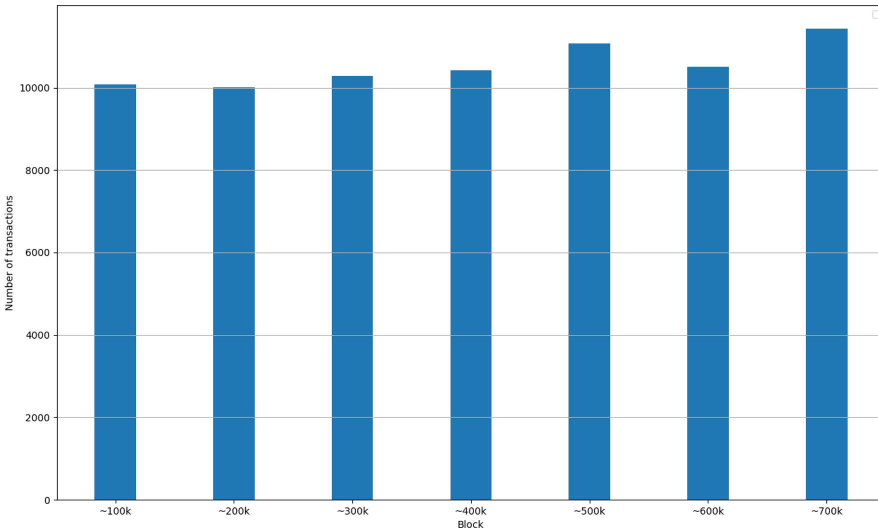


Figure 2.2: Transactions in data set 1. Total number of transactions is 73,798

- **Block ∼100,000 (Dec. 2010):** block 98,827 - block 101,174 (2348 blocks)
- **Block ∼200,000 (Sep. 2012):** block 199,962 - block 200,039 (78 blocks)
- **Block ∼300,000 (May 2014):** block 299,980 - block 300,021 (42 blocks)
- **Block ∼400,000 (Feb. 2016):** block 399,998 - block 400,003 (6 blocks)

- **Block ~500,000 (Dec. 2017):** block 499,999 - block 500,002 (4 blocks)
- **Block ~600,000 (Oct. 2019):** block 599,998 - block 600,002 (5 blocks)
- **Block ~700,000 (Sep. 2021):** block 699,993 - block 700,008 (16 blocks)

The number of Bitcoin transactions processed by the blockchain daily has been steadily increasing up to around 250,000 in 2021 [48]. While *data set 1* has a relatively small sample size, it contains a collection of transactions from different parts of Bitcoin history, which is interesting when reviewing analytics performed on it. Therefore, the second data set contains a larger number of sequential transactions - all transactions performed on one specific date.

February 1, 2022 was chosen as the date, resulting in *data set 2*: 271,146 transactions from block 721,253 to block 721,406.

- **(Feb. 1, 2022):** block 721,253 - block 721,406 (144 blocks)

The scripts for generating the data sets are explored in detail in section 4.2.

Chapter 3

Theory background

3.1 The structure of Bitcoin transactions

3.1.1 Bitcoin Client

Bitcoin clients are software implementations of the Bitcoin protocol, each having a complete copy of the blockchain [49]. Bitcoin clients are responsible for private key generation and verification and broadcasting of transactions. The Bitcoin system runs on a peer-to-peer network where each node is a device running a Bitcoin client [50].

3.1.2 Bitcoin Wallet

Bitcoin Wallets are lightweight software used to store private keys, giving users access to the keys that can be used to sign a transaction and spend available Bitcoins (BTC) [9].

3.1.3 Outputs

Outputs are the instructions for spending Bitcoin, containing a recipient address and a value. A transaction can have multiple outputs with arbitrary values set by the transaction's sender, but the total output value cannot exceed the combined value of the inputs [9].

3.1.4 Inputs

Inputs are references to unspent outputs [9]. As illustrated in Figure 3.1, the input of each transaction is linked to the output received from another transaction, resulting in the input having the same value and address as the referenced output. As Bitcoin input values can only be divided by being spent [31], the total input value (minus the *transaction fee*) must be spent in a transaction. If a sender wants to spend less in a transaction than the input value, the remaining value must be sent back to the

sender in the form of a *change* output. Transaction T_4 in Figure 3.1 is an example of this, where the sender has 0.5 BTC to spend, but he only wants to send 0.3 BTC to $Addr_F$. 0.02 BTC is spent as a fee, and 0.18 is sent back to the sender on $Addr_G$ (which the sender also controls). More about this in section 3.2.2.

3.1.5 Transaction fees

A *transaction fee* is a compensation value that is included in most transactions. It is given to miners who perform the work to confirm transactions on the public blockchain [9]. Lager fees result in a bigger incentive for a miner to include the transaction in a block, while a small fee might result in a delayed transaction. Mathematically, the *transaction fee* is the difference between the input and output values, as shown in Figure 3.1.

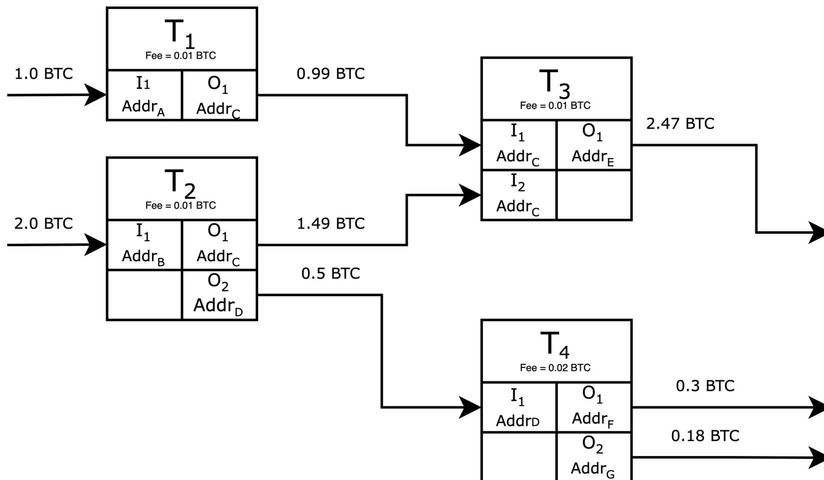


Figure 3.1: Bitcoin transactions including fees

3.2 Bitcoin Address clustering

Address clustering is the process of linking multiple addresses controlled by one user to one entity based on information from the blockchain [31]. Controlling an address is the ability to use this address as an input in a Bitcoin transaction, which is achieved by having access to the address's corresponding private key. Address clustering can, in many cases, have false positive or false negative results, defined as the following [13]:

False positive Include addresses that are not controlled by the same user in the same cluster.

False negative Not include addresses controlled by the same user in the cluster.

From a legal perspective, a false positive case means that an algorithm falsely claims that two addresses belong to the same entity. On the other hand, a false negative case is when the algorithm misses identifying that two different addresses belong to the same entity.

3.2.1 Common-input-ownership

One of the core methods for address clustering is the *common-input-ownership heuristic* (also called *multi-input heuristic* or *co-spent heuristic*). If one transaction has multiple input addresses, it can be assumed that they all belong to the same user [31].

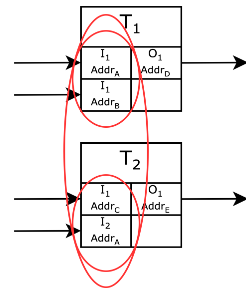


Figure 3.2: Common-input-ownership

A sender using an input address is required to have access to its corresponding private key and can therefore be considered the owner of the address. As addresses are reused, the number of entities can be reduced significantly. For example, if you have two transactions T_1 and T_2 with input addresses $Addr_A$, $Addr_B$ and $Addr_A$, $Addr_C$ respectively, not only are the input addresses in the individual transactions linked, but they also share the input address $Addr_A$. In other words, all of the addresses can be linked to the same entity in this case. This is illustrated in Figure 3.2.

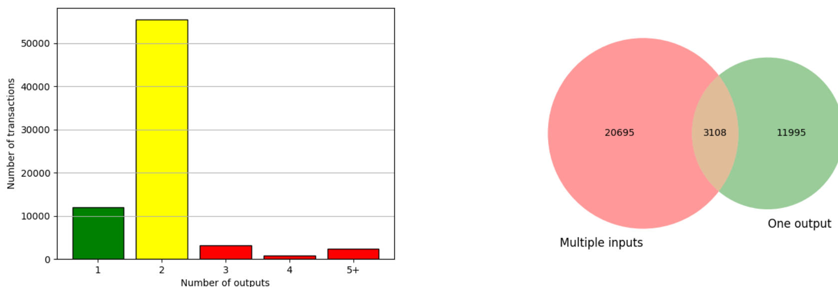
Existing heuristics

Heuristic 1.1 (Nakamoto, Satoshi) If two or more addresses are inputs of the same transaction, then all these addresses are controlled by the same user.

This is the original *common-input-ownership heuristic*, and the one used by Graphsense for its address clustering (see section 2.1). This heuristic has always been a known privacy issue in Bitcoin. It was first mentioned in the Bitcoin whitepaper [1] released three months prior to the first release of Bitcoin (version 0.1). In this paper, it is stated that linking input addresses of multi-input transactions is said to always be true. This fact has later become obsolete as *coin mixing* services such as CoinJoin, designed to break this heuristic, have been released [13]. These services obfuscate transactions by combining multiple transactions into one *joint transaction*, resulting in a transaction with multiple input addresses owned by different users [51]. This leads to joining the cluster of two different users who have only participated in one joint transaction. This clustering method is therefore considered a false positive, leading to alternative heuristics being proposed to not involve these transactions in the clustering.

Heuristic 1.2 (Ermilov *et al.*) If two or more addresses are inputs of the same transaction with one output, then all these addresses are controlled by the same user.

Ermilov *et al.* [52] restricts the original heuristic by only involving transactions with one output, as a large number of multi-output transactions are joint transactions created by *coin mixing*. As shown in Figure 3.3, only $\sim 16\%$ of the transactions in data set 1 fulfill this requirement, and only $\sim 26\%$ of these are transactions with multiple inputs. Using Heuristic 1.2, input addresses will be linked with a much lower chance of including mixed transactions but with a significantly lower address reduction ratio.



(a) Number of outputs for transaction in data set 1.

(b) Single-output transactions with multiple inputs.

Figure 3.3: Occurrences of transactions with multiple inputs and one output.

Our proposed heuristic

Excluding multiple-output transactions is a valid approach for minimizing the number of wrongfully linked addresses due to *coin mixing*. However, it will greatly impact the number of address reductions that can be done. As shown in Figure 3.3a, most Bitcoin transactions have two outputs. One of these outputs will, in many cases, be a change output [13] - and not a product of *coin mixing*. These transactions have only one actual receiver, as the other output is received by the sender itself. Because of this, the following heuristic is proposed:

Heuristic 1.3 (our proposed heuristic) If two or more addresses are inputs of the same transaction with one or two outputs, then all these addresses are controlled by the same user. If the transaction has two outputs, one of these must be a change output for the assumption to be valid.

Finding the change address of a transaction, or even knowing if there is one, is not trivial. There are several proposed heuristics to address this challenge, but none of them are entirely accurate. Our approach is to check for a self-change address and use our proposed OTC heuristic 2.4, explored in section 3.2.2.

3.2.2 One-time Change Address

The Change Output

As explained in section 3.1.4, an input is a reference to an unspent output from a previously received transaction. However, the entire value must be spent in the transaction not to lose the remaining value in *transaction fees*. For a Bitcoin transaction to be valid, the sum of input values must be equal to or higher than the sum of output values. Therefore, if one wants to spend less BTC in a transaction than the value of the referenced output minus the fee, an additional output must be created, where the output address is one controlled by the sender - this is the change output. If no change output is used, the remaining value of the input will be spent as a fee, as shown in transaction T_1 in Figure 3.4.

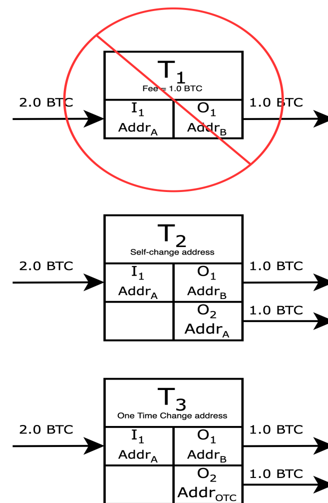


Figure 3.4: The change output

The value for the change output in the transaction sent back to the sender is the change

value, and the respective address is the change address. When conducting a transaction where change is required, the user could create a separate output for the remaining value and specify their own address as the output address. The address can be the same as the input address (self-change address) or a different one controlled by the sender. If this is not done manually by the user, a change output is created by the Bitcoin client with a generated address [13]. This address is now controlled by the user and is meant to be used only once as an input later. This is known as the One-Time Change (OTC) Address. Both variations of the change output are illustrated in transaction T_2 and T_3 in Figure 3.4.

Example scenario: Bob receives a payment from Alice of 2 BTC; Alice creates a transaction with an output with a value of 2 BTC and Bob's address as the output address. Change is not accounted for in this example.

```
1  T_1 =
2      Inputs:
3          I_1: { Address: Addr_Alice, Value: 2 }
4      Outputs:
5          O_1: { Address: Addr_Bob, Value: 2 }
```

Bob now has 2 BTC. He wants to share this with Carol, so he uses a Bitcoin client to create a new transaction with an output value of 1 BTC and Carol's address as the output address. However, Bob cannot set 1 BTC as input for this transaction since he must spend the entirety of the output sent from Alice. Therefore, the input must be the same as the output in Alice's transaction $T_1(O_1) = T_2(I_1)$ Bob uses 2 BTC as the input value and creates two outputs of 1 BTC - one for Alice and one for himself. Bob has two options when creating this transaction. The first is to use a self-change address, where he specifies his own address and remaining value as a change output in the transaction.

```
1  T_2 self-change =
2      Inputs:
3          I_1: { Address: Addr_Bob, Value: 2 }
4      Outputs:
5          O_1: { Address: Addr_Carol, Value: 1 }
6          O_2: { Address: Addr_Bob, Value: 1 }
```

This is a viable option; Bob receives the change value of 1 BTC back to his own address. However, now he has to use this same address as input for his next transaction as well, which promotes address reuse, which is not advisable [1]. He could also choose another address he controls that is not already included in the transaction as the change output address, but this quickly becomes tedious and chaotic when conducting multiple transactions. As mentioned, when the sum of the output values is lower than the value of the input minus the fee (which in this case

is zero), the Bitcoin client generates an OTC output. This is done automatically, and when Bob creates a transaction like this,

```
1  T_2 lower output value =
2      Inputs:
3          I_1: { Address: Addr_Bob, Value: 2 }
4      Outputs:
5          O_1: { Address: Addr_Carol, Value: 1 }
```

the Bitcoin client will automatically generate this transaction:

```
1  T_2 OTC =
2      Inputs:
3          I_1: { Address: Addr_Bob, Value: 2 }
4      Outputs:
5          O_1: { Address: Addr_Carol, Value: 1 }
6          O_2: { Address: Addr_OTC, Value: 1 }
```

where the OTC address ($Addr_{OTC}$), now controlled by Bob, is different from his original address and contains 1 BTC in value. He can now use this OTC output as an input in a new transaction.

Finding the OTC address

Using OTC addresses is the most common way of handling change, as it is done automatically and is considered safer in terms of privacy [13]. A self-change address will be linked to the user with the *common-input-ownership heuristic*. A user-specified change address is indistinguishable from an automatically generated OTC address if done correctly. The focus will therefore be on finding the OTC address.

Analyzing a transaction and distinguishing which output is for change and which is for spending is not easy. Finding the OTC addresses is more complicated than linking the inputs using the *common-input-ownership heuristic*. Not all transactions have change addresses (if the entirety of the value is spent), and some transactions can be mixed transactions with shared senders (explained in section 3.3.2)

Having a reliable way of distinguishing the change output from transactions, with as few false positives and, even more importantly, as few false negatives as possible, could have a significant impact on the privacy of Bitcoin. These addresses are often reused as inputs in later transactions, possibly in combination with other input addresses. By having a heuristic for finding the OTC address, combined with the *common-input-ownership heuristic*, the number of addresses can be reduced substantially and be a big factor in the process of deanonymizing Bitcoin transactions. This is explored further in section 3.2.3.

Existing heuristics

In this section, three existing heuristics for finding the OTC address are analyzed and used to propose a new one. All the heuristics share the following four properties:

(1) This is the first appearance of the one-time change output address $O(Addr_{OTC})$ on the blockchain
(2) The transaction is not a coin generation
(3) There is no address among the outputs that also appears in the inputs (self-change address)
(4) The transaction has exactly two outputs

Because the OTC address is automatically generated from the Bitcoin client, this is the first appearance of this address (1). Excluding transactions with self-change addresses (3) is done because there is no need for an OTC address in these cases, and the address will be clustered with the *common-input-ownership heuristic*. Only including transactions with two outputs (4) is adequate and makes it easier to determine the OTC address. This is because $\sim 90\%$ of transactions have either one or two outputs, as seen in Figure 3.3a. The reason for this is that transactions generated by consumer wallets always have one or two outputs [53]. Additional properties are mainly based on determining that the other output is not a change output. These properties vary, as more limiting factors will result in a lower occurrence of false positives but also a higher occurrence of false negatives.

Heuristic 2.1 (Meiklejohn *et al.*)

(1) This is the first appearance of the one-time change output address $O_{OTC}(Addr)$ on the blockchain
(2) The transaction is not a coin generation
(3) There is no address among the outputs that also appears in the inputs (self-change address)
(4) The transaction has exactly two outputs
(5) This is not the first appearance of the other output address $O_{other}(Addr)$ on the blockchain

Meiklejohn *et al.* [31] defines heuristic 2.1, where in addition to the properties mentioned, the address which is not the OTC address must have been used as either an input or output in a previous transaction for the assumption of the OTC address to be valid. This is based on the fact that the OTC address is generated for the specific transaction. Therefore, it is possible only for the other address to already exist on the blockchain.

Heuristic 2.2 (Zhang *et al.*)

(1) This is the first appearance of the one-time change output address $O_{OTC}(Addr)$ on the blockchain

(2) The transaction is not a coin generation

(3) There is no address among the outputs that also appears in the inputs (self-change address)

(4) The transaction has exactly two outputs

(5) Only the other output address $O_{other}(Addr)$ is reused as an output address in some later transaction

In response to Heuristic 2.1, Zhang *et al.* [13] proposes an alternative way of determining non-change addresses. Instead of only looking at previous use of the output addresses to determine this, their proposed heuristic checks for reuse of these addresses as outputs in later transactions (5). The OTC address should only be referenced once as an input address - so if an address is reused as an output in later transactions, it is not an OTC address. This approach contributes positively to the ratio of address reduction when combined with Heuristic 2.1 but could also lead to a higher degree of false positive results [13].

Heuristic 2.3 (Ermilov *et al.*)

(1) This is the first appearance of the one-time change output address $O_{OTC}(Addr)$ on the blockchain

(2) The transaction is not a coin generation

(3) There is no address among the outputs that also appears in the inputs (self-change address)

(4) The transaction has exactly two outputs

(5) This is not the first appearance of the other output address $O_{other}(Addr)$ on the blockchain

(6) The number of inputs in transaction is not equal to two

(7) $O_{other}(Addr)$ has not been OTC addressed in previous transactions

(8) The decimal representation of the change value $O_{OTC}(Value)$ has more than 4 digits after the dot

Ermilov *et al.*'s [52] approach to finding the OTC address goes in a different direction than Heuristic 2.2. Instead of trying to include more transactions in the results, possibly resulting in more false positives, this heuristic limits heuristic 2.1

by adding multiple factors, resulting in less positive results, both false and true. However, the number of false negatives will be significantly reduced.

Not including transactions with two inputs (6) is a property for excluding mixed transactions in the cluster. The number of outputs is two, and having the same number of inputs as outputs will often be caused by the transaction being a result of *coin mixing* [52]; therefore, there will be no change address that can be linked to one user. As seen in Figure 3.5, $\sim 15\%$ of the transactions in data set 1 have two inputs. Another new factor in this heuristic is (7) to check if the other output has been OTC addressed in any previous transactions. An OTC address is not reused, and eliminating these cases will reduce the number of false positives. A very limiting property in this heuristic is the determination of the validity of the OTC output based on if the value has more than four digits after the dot (8). This property is especially restrictive when analyzing transactions done early in the lifespan of Bitcoin and can result in many false negatives. Because earlier transactions involved much larger values due to the value of Bitcoin as a currency being much lower than it is today, these transactions often had much higher input values with a few decimals, leading to the change output value having the same property.

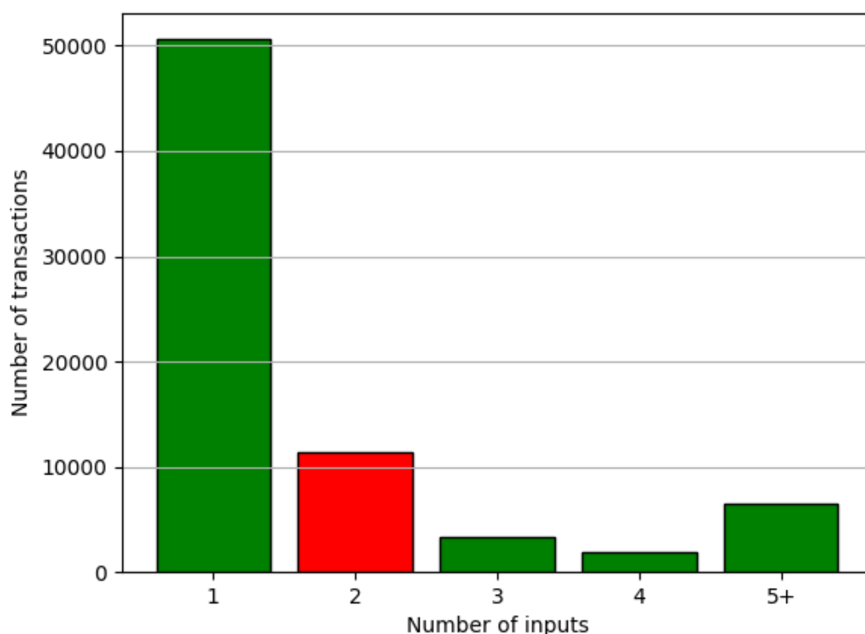


Figure 3.5: Number of inputs for transaction in data set 1

Our proposed Heuristic

Considering the three existing heuristics for determining the OTC address, an ideal heuristic that can achieve the optimal ratio of false positives and false negatives is proposed. This is also the heuristics used to determine if a transaction has an OTC address in H1.3.

Getting the lowest ratio of false negative results is already the purpose of Zhang *et al.* [13], who does combine the existing heuristic (H2.1) with his variation (H2.2) in an OR fashion. This approach will be referred to as H2.1+H2.2. However, as H2.1+H2.2 produces a significant amount of false positive results, the proposed heuristic adds two additional properties to H2.1+H2.2. Of the three additional properties of H2.3, only property (7) was included - the output address, which is not the change address, has not been used as an OTC address in any previous transactions. Property (8) is excluded because it leads to many false negative results, and property (6) because it will exclude too many multi-input transactions, which has a considerable impact on H1.3.

The *Optimal Change Heuristic* is included in our proposed heuristic, defined as the following [53]:

Optimal Change Heuristic (Nick, Jonas David) The one-time change value $O_{OTC}(Value)$ is smaller than any of the inputs.

If the change value is higher than any of the input values, then the input would not be necessary for the transaction and would only result in additional *transaction fees* [51].

These additional properties should reduce the number of false positives considerably while maintaining a sufficient address reduction ratio. Our proposed heuristic is defined as the following.

Heuristic 2.4 (our proposed heuristic)

(1) This is the first appearance of the one-time change output address $O_{OTC}(Addr)$ on the blockchain

(2) The transaction is not a coin generation

(3) There is no address among the outputs that also appears in the inputs (self-change address)

(4) The transaction has exactly two outputs

(5) This is not the first appearance of the other output address $O_{other}(Addr)$ on the blockchain OR only the other output address $O_{other}(Addr)$ is reused as an output address in some later transaction

(6) $O_{other}(Addr)$ has not been OTC addressed in previous transactions

(7) The one-time change value $O_{OTC}(Value)$ is smaller than any of the inputs.

3.2.3 Combination of common-input-ownership heuristic and OTC heuristic

The impact of finding the change address of a transaction becomes clear when evaluating it in combination with the *common-input-ownership heuristic*. Finding the change addresses of transactions only adds one additional address to the entity's cluster. However, if these addresses reside in different clusters created by a *common-input-ownership heuristic*, it could lead to the joining of clusters. As illustrated in Figure 3.6, because the output address used as a change address in T_1 is reused as an input address in T_2, all of the input addresses can be linked to the same cluster, even though the transactions share no *common-input* addresses.

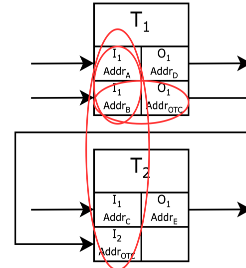


Figure 3.6: Combination of common-input-ownership and OTC

3.3 Difficulties in performing address clustering

This section will discuss methods that make it harder to perform address clustering on transactions. The intentions of our proposed heuristics in section 3.2 are to skip these transactions when performing clustering. This was the best approach for avoiding false positive results, as most Bitcoin transactions are conducted without the use of obfuscation techniques [54]. Only anonymity enhancements conducted on the original Bitcoin protocol are discussed in this section, not forks of Bitcoin or other cryptocurrencies with enhanced privacy like Zcash.

3.3.1 Unique key pairs

As stated in the original Bitcoin paper, a new key pair should be used for each new transaction to prevent them from being linked to a common owner [1]. Many Bitcoin users follow this recommendation, which is visible in the number of new addresses generated each month compared to the number of transactions [12].

3.3.2 Coin mixing

Coin mixing is a common strategy used in cryptocurrency transactions to enhance anonymity, creating challenges for the clustering heuristics discussed in section 3.2. A *coin mixer* will obfuscate cryptocurrency transactions in a way that complicates tracing the flow of payments on the blockchain. *Coin mixing* is possible on any cryptocurrency with a public ledger [55]. Because of the effectiveness of *coin mixing* and the way it enhances anonymity, there have been discussions of potentially implementing certain *coin mixers* directly on top of the Bitcoin core [8]. This change has not been implemented, and the volume of mixed transactions is still small. Therefore, regular address clustering algorithms are still effective on the vast majority of Bitcoin transactions.

There are two kinds of *coin mixers*, centralized and decentralized [55]:

Centralized mixer Also known as tumblers, centralized mixers are single entity mixing servers that receive deposits from multiple participants and either return or forward the value from a different participant's address disassociated from the sender's original address [55]. This is illustrated in Figure 3.7. Some mixers will also add delays or split the original value into separate transactions to make the mixing harder to detect [56, 16]. A mixing server will typically take a mixing fee of around 1-3% of the total Bitcoin value [57]. A concern with centralized mixers is that they break the decentralization of Bitcoin transactions and learn about the relation between the input and output addresses [58]. There is also no guarantee that the service returns the value to the participant. Therefore, the anonymity and reliability of such mixers heavily rely on the assumption that the mixer does not log or reveal information about address relations and lawfully returns the original value [58]. These types of mixing services are provided by *coin mixing* companies such as Blender.io, ChipMixer, and FoxMixer [59].

Decentralized mixer Based on protocols like CoinJoin [55], decentralized mixers create combined transactions with multiple users participating using one or more input addresses. The output addresses and output values are shuffled, obfuscating the details of the transactions, making them very difficult to distinguish from one another [8]. The CoinJoin protocol is explored further in section 3.3.3.

3.3.3 CoinJoin

CoinJoin is an open-source *coin mixing* protocol proposed in 2013 by Gregory Maxwell [8]. As opposed to centralized mixing servers, CoinJoin is a decentralized, trustless mixing protocol that does not require a third-party entity. CoinJoin is built on the

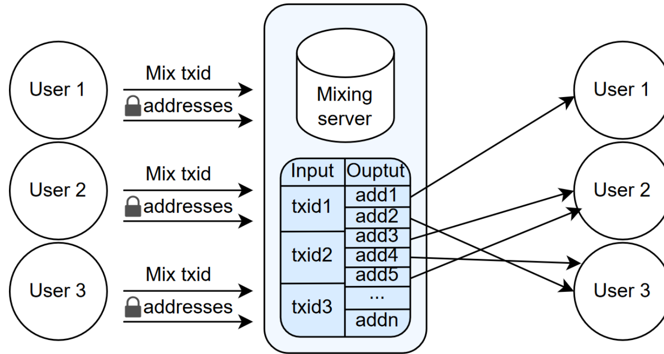


Figure 3.7: Illustration of a centralized coin mixing scheme

principle of combining multiple transactions into a single combined transaction [8]. By being a decentralized protocol, CoinJoin eludes the main issues of using single entity centralized mixing servers - revealing sensitive information to a third party, and trusting the service with proper handling of the funds.

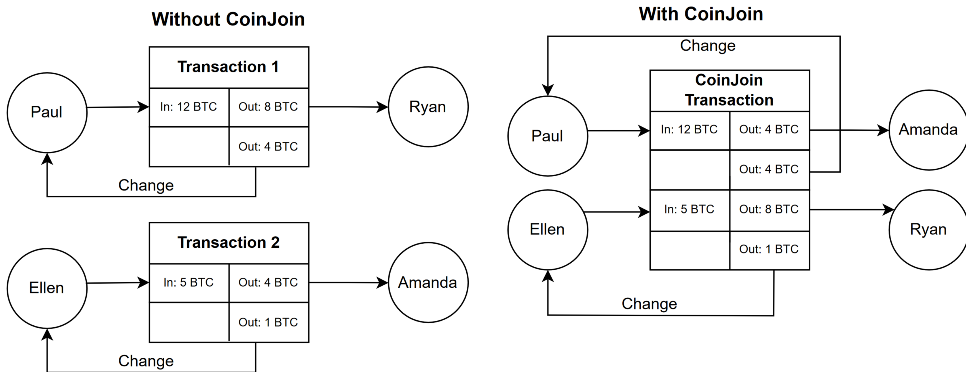


Figure 3.8: Illustration of the concept behind CoinJoin

Figure 3.8 illustrates how CoinJoin works. Transaction 1 and transaction 2 are separate transactions, vulnerable to the change address heuristics described in section 3.2.2. Combining them into a CoinJoin transaction makes it much harder to distinguish these transactions, especially when the set of participants is large. Participants can use chat services to plan the details of CoinJoin transactions before it is conducted [8]. A concern with this method is that participants will learn the other participants' address relations and reveal them to the chat service. This is why CoinShuffle was proposed in 2014, which uses ring signatures to hide amounts and addresses from other participants [58]. Participants can also perform DoS attacks against CoinJoin transactions, i.e., by refusing to sign the transaction or spending

the input before the transaction is completed. However, excluding such participants in future transactions solves this problem [8].

Identifying CoinJoin transactions

There are some key characteristics of CoinJoin transactions compared to regular transactions, the most obvious being multiple inputs and outputs. However, a multi-input/output transaction is not necessarily the result of CoinJoin, as this could be the case in any regular transaction. CoinJoin transactions do, however, have specific characteristics regarding the individual amounts transferred to each address [60]. As a result, the sums of the respective input and output subsets will usually correspond with each other and is considered a significant weakness of both CoinJoin and CoinShuffle [61].

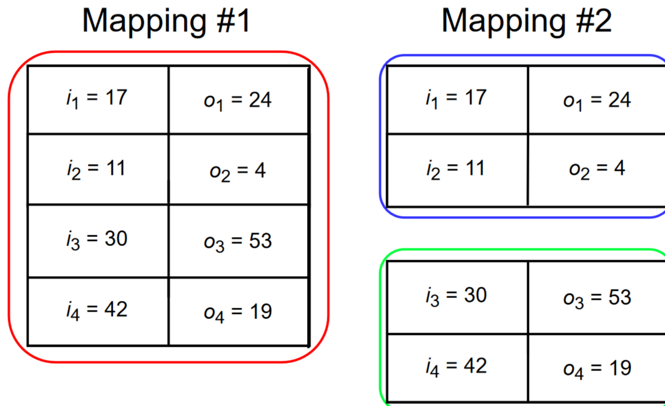


Figure 3.9: Example of sub-transactions within a CoinJoin transaction

Looking at Figure 3.9, Mapping #1 represents what looks like a regular transaction, with a total value of 100 BTC transmitted. Looking closer at the individual input and output values however, it is observed that the sum of inputs i_1 and i_2 are equivalent to the sum of outputs o_1 and o_2. This way, i_1 and i_2 can be grouped with o_1 and o_2 as a sub-transaction [60]. The same can be done for i_3, i_4, o_3 and o_4. Finding sub-transactions in this manner makes a strong case for a transaction being the product of CoinJoin. It is, however, impossible to know how many sub-transactions are in a CoinJoin transaction, because some sub-transactions may also be owned by the same user. However, one output address will never be part of more than one sub-transaction [60].

The first part of identifying sub-transactions is finding the subset sums of the inputs and outputs, which is an NP-complete problem [62]. The number of possible

subset sums is 2^N, where N is the number of addresses in one CoinJoin transaction [63].

Algorithm 3.1 is an approach written in Python to find all possible subset sums in a list of values. This algorithm must be run on the set of inputs and outputs individually; then, the respective sums are compared. Even though the algorithm is NP-complete, it may be feasible to run this algorithm on a large set of transactions. This is because the asymptotic running time depends only on the number of addresses, which in most cases are too few to create a bottleneck with a significant delay. Also, this algorithm does not consider *transaction fee*, which is often low and split equally among the participants. However, this is impossible to determine. These issues are discussed more in section 6.4.

Algorithm 3.1: Find all possible subset sums in a list of values

```
1  def subsetSums(values):
2      n = len(values)
3      possible_sums = []
4
5      total_subsets = 2**n
6
7      for i in range(1, total_subsets):
8         Sum = 0
9         sum_indexes = []
10        # Consider binary representation of current
11        # i to decide which elements to pick.
12        for j in range(n):
13           if ((i & (1 << j)) != 0):
14               print(values[j])
15               sum_indexes.append(j)
16               Sum += values[j]
17        possible_sums.append([Sum, sum_indexes])
18     return possible_sums
```

Chapter 4
Implementation details

4.1 Initial setup

4.1.1 Python

The Python workspace was set up with a virtual environment and environmental variables. As the virtual environment was used to create an isolated environment, only the packages explicitly needed for this project were installed. The virtual environment was set up and activated with the following commands:

```
$ python3 -m venv myvenv
$ . myvenv/bin/activate
```

Environmental variables were used to hide sensitive data on Github, such as the Graphsense API key and MongoDB connect URI. It was also very beneficial when executing the scripts on the Virtual Machine, where a different MongoDB connect URI with "localhost" as the hostname had to be used. The environmental variables were placed in a .env file (not pushed to Github) and accessed using *python-dotenv*, as shown in Algorithm 4.1.

Algorithm 4.1: Accessing the API key's environmental variable

```
1 from dotenv import load_dotenv
2 import os
3
4 load_dotenv('.env')
5
6 api_key = os.environ.get("api_key")
```

4.1.2 Graphsense

The clustering heuristics discussed in section 3.2 were implemented using the Graphsense Python Client, which was installed via pip and set up by following the setup procedure from the client's documentation [41]. The Graphsense Python client interacts with the Graphsense REST API.

The Graphsense Python Client consists of eight API classes: *AddressesApi*, *BlocksApi*, *BulkApi*, *EntitiesApi*, *GeneralApi*, *RatesApi*, *TagsApi*, *TxsApi*

An example of how the client with an API class is imported, configured, and used to perform a simple request is shown in Algorithm 4.2.

Algorithm 4.2: Graphsense python client example - transactions for block with height 500,000

```
1  import graphsense
2  from graphsense.api import blocks_api
3  import os
4
5  api_key = os.environ.get("api_key")
6  configuration =
       graphsense.Configuration(host="https://api.graphsense.info")
7  configuration.api_key["api_key"] = api_key
8
9  api_client = graphsense.ApiClient(configuration)
10
11 blocks_api = blocks_api.BlocksApi(api_client)
12
13 transactions_for_block = blocks_api.list_block_txs('btc', 500000)
```

As mentioned in section 2.1, the API was limited to 1,000 requests per hour, severely affecting the runtime. A workaround for this was discovered by using the bulk request feature of the API, which could perform an arbitrary number of sub-requests in one big request, mitigating the impact on the request limit. As the bulk requests have a response time proportional to the number of sub-requests done, the number of requests included in the bulk requests was adjusted accordingly to reach 1,000 requests per hour. There are no regulations in Graphsense's terms of use regarding the use of bulk requests in this manner [64]. However, due to their limited server capacity, we were subsequently excluded from using this feature. By contacting the Graphsense team and promising to use the feature more sparingly, we were re-granted access, and future bulk requests were executed with a maximum of 50 sub-requests. Later Graphsense updated its platform to include all sub-requests of a bulk request in the request ratio. However, at this time, our request limit was already temporarily increased to 10,000.

4.1.3 Virtual Machine

The Virtual Machine running Ubuntu 20.04.3 was accessed with SSH and used for running scripts and storing data in a MongoDB database. The scripts were pulled from a Github repository and executed without hangups using the nohup linux command [65]. The nohup command, shown below, allows commands to be executed without being interrupted when closing the remote connection.

```
$ nohup python -u aggregate_outputs.py &> log.out &
```

MongoDB

A MongoDB database was set up and hosted on the VM, which led to extremely low response time when inserting documents and executing queries compared to using a hosting service. The PyMongo package was used in Python to interact with the database. An example of querying and inserting documents using PyMongo is shown in Algorithm 4.3.

Algorithm 4.3: Pymongo example - query for transactions with multiple inputs and insert into a new collection

```
1  import pymongo
2  from dotenv import load_dotenv
3  import os
4
5  load_dotenv('.env')
6
7  connectURI = os.environ["connectURI"]
8  client = pymongo.MongoClient(connectURI)
9
10 db = client["master"]
11 collection_1 = db['transactions']
12 collection_2 = db['transactions_multiple_inputs']
13
14 transactions_with_multiple_inputs = collection_1.find({ "$expr": {
       "$gt": [{"$size": "$inputs"}, 1]}})
15 collection_2.insert_many(transactions)
```

4.2 Data sets

As discussed in section 2.5, two data sets were used; *data set 1* with a selection of $\sim 10,000$ transactions from various parts of Bitcoin's history, and *data set 2* with a larger set of transactions from an arbitrary date - February 1, 2022.

For both data sets, a list of relevant blocks was found. Then all the transactions of the blocks were retrieved using Graphsense REST API via the Graphsense Python Client and inserted into the MongoDB database. The complete script for generating

the data sets is shown in Appendix C.1, while the most significant implementations are covered in this section.

The combined number of blocks for both data sets is 2,643, with a total of 344,944 transactions. All the transactions for one block were retrieved from Graphsense using a single request, resulting in a runtime of under 3 hours when conducting 1,000 requests per hour. However, as the inputs and outputs were not included in this request, they had to be requested separately and subsequently included in the transactions. Therefore, two requests had to be performed for each transaction; one for the inputs and one for the outputs. With 344,944 transactions, this would mean a total of almost 700,000 requests, leading to a runtime of over 29 days. A workaround using bulk requests was implemented where the inputs or outputs of 50 transactions were retrieved in each request, as shown in Algorithm 4.4. Fortunately, this was implemented and executed before each sub-request in a bulk request impacted the hourly request limit, enabling a reduced runtime of 33 hours. One problem with this approach was that the bulk requests returned a list of inputs or outputs for 50 different transactions, which then had to be distinguished and inserted into the correct transaction.

Algorithm 4.4: Get all transactions from a block with inputs and outputs

```
1  def get_io(tx_hashes, io):
2      io_list = []
3      i = 0
4      while i < len(tx_hashes):
5          try:
6              body = {
7                  "tx_hash": tx_hashes[i:i+50], "io": io}
8              io_list.extend(bulk_api.bulk_json(
9                  'btc', 'get_tx_io', 1, body, async_req=True).get())
10         except graphsense.ApiException as e:
11             # Request limit exceeded
12             if (e.status == 429):
13                 sleep(int(e.headers["Retry-After"]) + 60)
14                 continue
15             else:
16                 raise e
17         i += 50
18     return io_list
19 def get_transactions_from_block_with_io(block_height):
20     transactions = blocks_api.list_block_txs('btc', block_height)
21     tx_hashes = [transaction["tx_hash"] for transaction in
           transactions]
22     inputs = get_io(tx_hashes, "inputs")
23     outputs = get_io(tx_hashes, "outputs")
24     return format_transactions(transactions, inputs, outputs)
```

4.2.1 Data set 1

For *data set 1*, intervals of blocks from every 100,000 blocks on the Bitcoin blockchain were used, with the condition that they were the nearest blocks containing at least 10,000 transactions in total.

Algorithm 4.5 takes a block height as input and expands a list containing the surrounding blocks until the total transaction count reaches at least 10,000. The list of relevant block heights is subsequently sorted and returned. This algorithm was used on block heights [100000, 200000, ..., 700000] in which all the transactions for each block were retrieved using the *Blocks_api.list_txs* method and Algorithm 4.4 before being inserted into a MongoDb collection using Pymongo's *insert_many* method.

Algorithm 4.5: Given a block height, find surrounding blocks to reach desired number of transactions

```
1  def get_list_of_surrounding_blocks_for_number_of_transactions(
       block_height, number_of_transactions):
2      current_number_of_transactions = 0
3      current_block_height = block_height
4      blocks = []
5      increment = 1
6      while current_number_of_transactions < number_of_transactions:
7          blocks.append(current_block_height)
8          number_of_transactions_in_block = blocks_api.get_block('btc',
               current_block_height)["no_txs"]
9          current_number_of_transactions +=
               number_of_transactions_in_block
10         current_block_height = block_height + increment
11         increment = -increment if increment > 0 else -increment + 1
12     return sorted(blocks)
```

4.2.2 Data set 2

For *data set 2*, all the blocks for a given time interval were required. Given that the Bitcoin blocks are chronologically ordered [47], only the first and last block processed in the time interval had to be discovered to get a list of all the blocks in between. However, finding the first and last blocks of the interval was not trivial, as the Graphsense API has no endpoint for getting a block closest to a given timestamp.

Graphsense's *BlocksApi* had an endpoint for getting some data for a given block height; *blocks_api.get_block*. This data included the timestamp for a given block. Since all blocks were sorted chronologically, a variation of the binary search algorithm [66] was implemented to find the relevant block. Algorithm 4.6 takes a timestamp, a minimum block height (zero), a maximum block height (the height of the latest Bitcoin block in Graphsense), and an *ensuing* condition as inputs. A recursive

approach to binary search is used to return the closest block processed before or after the given timestamp based on the *ensuing* condition.

Algorithm 4.6 was used with the *ensuing* condition set to *True* for the start date and *False* for the end date, resulting in a *start_block* and an *end_block*, which are used to generate a list of all the block heights in the time interval.

Algorithm 4.6: Find the Bitcoin block closest to a given timestamp

```
1  def find_block_by_timestamp_binary_search(min_block_height,
        max_block_height, timestamp, ensuing):
2      if max_block_height >= min_block_height:
3          mid_block_height = (max_block_height + min_block_height) // 2
4          mid_block = blocks_api.get_block('btc', mid_block_height)
5          if mid_block['timestamp'] == timestamp:
6              return mid_block_height
7          elif mid_block['timestamp'] > timestamp:
8              return
                   find_block_by_timestamp_binary_search(min_block_height,
                   mid_block_height - 1, timestamp, ensuing)
9          else:
10             return
                   find_block_by_timestamp_binary_search(mid_block_height
                   + 1, max_block_height, timestamp, ensuing)
11     else:
12         if ensuing:
13             return min_block_height
14         else:
15             return max_block_height
```

4.3 Common-input-ownership heuristics

The *common-input-ownership* heuristics discussed in section 3.2.1 were implemented using two algorithms that had to be executed in sequence. The complete script is shown in Appendix C.2.

Slow runtime was an issue when executing these algorithms on the data sets. As the list of entities and the number of addresses in each entity's cluster grew, the query time got significantly higher. However, by creating a MongoDB multikey index [67] for each *address_cluster* array, the total runtime was reduced by over 60 percent.

Algorithm 4.7 was used on transactions where the conditions for assuming a shared owner of the inputs were met. For Heuristic 1.1, this is the case for all transactions. However, for Heuristic 1.2 and Heuristic 2.3, this is based on the number of outputs and the occurrences of a change address in the transactions, detailed in section 3.2.1. Algorithm 4.7 populates an empty collection with entities from a list of transactions. The algorithm iterates through all the transactions and checks for an existing entity

containing one or more input addresses of a transaction. If an entity is found, the additional input addresses are linked to the entity. If not, a new one is created.

Algorithm 4.7: Create entities with shared input addresses from a list of transactions

```
1  def common_input_address_clustering(transactions, entities_collection):
2      entities_collection.create_index("address_cluster")
3      for transaction in tqdm(transactions):
4          input_addresses = get_addresses(transaction["inputs"])
5          existing_entity = False
6          for index, address in enumerate(input_addresses):
7              #Try to find entity with address in cluster
8              entity_with_shared_input_address =
                      entities_collection.find_one({"address_cluster":
                      address})
9              if entity_with_shared_input_address:
10                 existing_entity = True
11                 entity_id = entity_with_shared_input_address['_id']
12                 entities_collection.update_one({ '_id': entity_id }, {
13                     '$addToSet': { 'address_cluster': { '$each':
                          input_addresses}},
14                     '$push': {'tx_hashes': transaction["tx_hash"]}})
15                 input_addresses = input_addresses[index+1:]
16                 # check for existing entitites with any of the input
                      addresses in cluster and combine them with this
                      one
17                 for input_address in input_addresses:
18                     exiting_entity_shared_address =
                          entities_collection.find_one({"$and":
                          [{"address_cluster": input_address}, {"_id":
                          {"$ne":
                          entity_with_shared_input_address['_id']}}]})
19                     if exiting_entity_shared_address:
20                         entities_collection.update_one({ '_id':
                              entity_id }, {
21                             '$addToSet': { 'address_cluster': {
                                  '$each':
                                  exiting_entity_shared_address[
                                  "address_cluster"]}},
22                             '$push': {'tx_hashes': { '$each':
                                  exiting_entity_shared_address[
                                  "tx_hashes"]}}})
23                         entities_collection.delete_one({"_id":
                              exiting_entity_shared_address["_id"]})
24                 break
25         if not existing_entity and len(input_addresses) > 0:
26             entities_collection.insert_one({"address_cluster":
                  list(dict.fromkeys(input_addresses)), "tx_hashes":
                  [transaction["tx_hash"]]})
```

For heuristic 1.2 and 2.3, the transactions that do not meet the conditions are assumed to have one entity for each input address. For these transactions, Algorithm 4.8 is run subsequent to Algorithm 4.7. An entity is then created for each input

address that is not part of an existing entity.

Algorithm 4.8: Create entities for each input address from a list of transactions

```
1 def individual_input_address_clustering(transactions,
      entities_collection):
2     entities_collection.create_index("address_cluster")
3     for transaction in tqdm(transactions):
4         input_addresses = get_addresses(transaction["inputs"])
5         for input_address in input_addresses:
6             if not entities_collection.find_one({"address_cluster":
                   input_address}):
7                 entities_collection.insert_one({"address_cluster":
                       [input_address], "tx_hashes":
                       [transaction["tx_hash"]]})
```

4.4 One-time change address heuristics

The four OTC heuristics discussed in section 3.2.2 were implemented using several algorithms developed through trial and error to minimize the number of requests made to the Graphsense API. The most significant parts of the implementation are covered in this section; the complete script is shown in Appendix C.3.

The execution order of the algorithms is determined by which inputs they require and the number of requests they make to the Graphsense API. Some algorithms require information returned by other algorithms, such as the determined OTC address candidate; these have to be executed late in the implementation, regardless of their request limit consumption.

4.4.1 Coin generation and the number of inputs and outputs

Properties (2) *the transaction is not a coin generation,* (4) *the transaction has exactly two outputs* and H2.3(6) *the number of inputs in the transaction is not equal to two* are all based on information already included in the transaction data retrieved from the data set. These properties are the first to be checked because if they are false, the transaction can be determined not to have an OTC address without using any of the request limits. The implementation of this approach is shown in Algorithm 4.9.

Algorithm 4.9: Coin generation, number of inputs and number of outputs

```
1 def is_coin_generation(transaction):
2     return transaction['coinbase']
3
4 def has_two_outputs(transaction):
5     return len(transaction["outputs"]) == 2
6
7 def has_two_inputs(transaction):
8     return len(transaction["inputs"]) == 2
```

4.4.2 Occurrence of self-change address

Algorithm 4.10 determines the occurrence of a self-change address in a transaction by iterating through all the output addresses and checking for a matching address in the input addresses. Because this information is already included in the transaction from the data set collection, this does not perform any requests to the Graphsense API either.

Algorithm 4.10: Check for self-change address in the transaction

```
1 def has_self_change_address(transaction):
2     input_addresses = get_addresses(transaction["inputs"])
3     output_addresses = get_addresses(transaction["outputs"])
4     for output_address in output_addresses:
5         if output_address in input_addresses:
6             return True
7     return False
```

4.4.3 Address reused as output

Since OTC addresses are only to be used once as an output, Algorithm 4.11 was implemented to check that this was the case for the OTC address but not for the other output address - the condition for Heuristic 2.2. The algorithm finds the last transaction where the address is used as an output; it then checks if it is the same transaction or not. Graphsense's *Addresses_api.list_address_txs* function is used in the implementation to retrieve a list of all the transactions in which an address has been involved; this had to be done using multiple requests. The list returned from Graphsense is in reverse chronological order. However, the transactions where the address has been used as an input are listed before the transactions where it has been used as output. Therefore, the algorithm must first request all the address's input transactions before requesting its output transactions. Algorithm 4.11 does this by skipping all transactions until an output transaction is found and then using binary search on the newest part of the list to find the address's last

output transaction. The transaction hash of this transaction is then compared with the original transaction hash. For addresses involved in thousands of transactions, this was extremely demanding on our request limit and created a bottleneck when executing the implementation of the heuristics. If Graphsense's API were to have a parameter for this function, where the transactions the address has been involved in could be filtered based on if it was part of the inputs or outputs, the execution time could be significantly reduced. Several variations of Algorithm 4.11 were tested, but this implementation performed the best.

Algorithm 4.11: Check if an output is reused as an output in a later transaction

```
1  def get_latest_output_transaction_binary_search(transactions, low,
       high):
2      if high >= low:
3          mid = (high + low) // 2
4          if transactions[mid].value.value > 0 and
                transactions[mid-1].value.value < 0:
5              return transactions[mid]["tx_hash"]
6          elif transactions[mid].value.value > 0:
7              return get_latest_output_transaction_binary_search(
                   transactions, low, mid - 1)
8          else:
9              return get_latest_output_transaction_binary_search(
                   transactions, mid+1, high)
10     else:
11         return False
12
13 def is_used_as_output_later(address, current_transaction_tx_hash):
14     response = addresses_api.list_address_txs(
15         'btc', address, pagesize=500)
16     if response["address_txs"][0]["tx_hash"] ==
            current_transaction_tx_hash:
17         return False
18     while response:
19         transactions = response["address_txs"]
20         if transactions[-1].value.value > 0:
21             return get_latest_output_transaction_binary_search(
                   transactions, 0, len(transactions)-1) !=
                   current_transaction_tx_hash
22         response = addresses_api.list_address_txs('btc', address,
                page=response['next_page'], pagesize=500)
```

4.4.4 Number of digits in OTC value

Algorithm 4.12 checks if an OTC value has more than four digits after the dot by converting the value of the transaction, which is in satoshi, to BTC.

$$BTC = satoshi * 10^{-8}$$

The value is then converted to a string and split at the decimal point. The length of the latter part of the string is then checked.

Algorithm 4.12: Check if the decimal representation of the OTC value has more than four digits after the dot

```
1 def value_has_more_than_four_digits_after_dot(value):
2     value = '{0:.8f}'.format(value * 10**(-8)).strip("0")
3     number_of_decimals = len(value.split(".")[1])
4     return number_of_decimals > 4
```

4.4.5 Optimal change heuristic

To check if the optimal change heuristic is valid for a transaction, Algorithm 4.13 compares the OTC value with all the input values and returns *False* if any of the input values are smaller than the change value.

Algorithm 4.13: Optimal change heuristic

```
1 def otc_value_is_smaller_than_all_input_values(otc_value, inputs):
2     input_values = [inp["value"]["value"] for inp in inputs]
3     for input_value in input_values:
4         if input_value < otc_value:
5             return False
6     return True
```

4.4.6 Determining the OTC output

Determining which of the two outputs is the OTC output and which is not was done by checking for the properties in the heuristics that defined the OTC address - if it was its first appearance (Heuristics 2.1, 2.3 and 2.4) or if it was the only address not reused as an output (Heuristics 2.2 and 2.4). The implementation of this is shown in Algorithm 4.14. This enabled the algorithms used on the data specifically for the OTC output or the other output to be executed, as this information was now available.

Algorithm 4.14: Determining the OTC output

```
1    is_first_transaction_of_output_address_1 =
         get_first_transaction_hash(output_1["address"][0]) ==
         transaction['tx_hash']
2    is_first_transaction_of_output_address_2 =
         get_first_transaction_hash(output_2["address"][0]) ==
         transaction['tx_hash']
3
4   # (1) This is the first appearance of the OTC address;
5    if is_first_transaction_of_output_address_1 ==
         is_first_transaction_of_output_address_2 == False:
6        return otc_data
7
8    # (5) This is not the first appearance of the other output address
         OR only the other output address is reused as an output
         address in some later transaction
9    elif is_first_transaction_of_output_address_1 ==
         is_first_transaction_of_output_address_2 == True:
10       output_1_used_later =
             is_used_as_output_later(output_1["address"][0],
             transaction["tx_hash"])
11       output_2_used_later =
             is_used_as_output_later(output_2["address"][0],
             transaction["tx_hash"])
12       if output_1_used_later == output_2_used_later:
13           return otc_data
14       else:
15           if output_1_used_later:
16               otc_output = output_2
17               other_output = output_1
18           elif output_2_used_later:
19               otc_output = output_1
20               other_output = output_2
21   else:
22       if is_first_transaction_of_output_address_1:
23           otc_output = output_1
24           other_output = output_2
25       elif is_first_transaction_of_output_address_2:
26           otc_output = output_2
27           other_output = output_1
```

4.5 Combination of common-input-ownership heuristic and OTC heuristic

If a change output is discovered in a transaction, this can be used to combine entities when the address is reused. The implementation of this is shown in Algorithm 4.15, which achieves this by finding other entities containing a change address found by Heuristic 2.4 and combining them with the original ones.

Algorithm 4.15: Reduce the clusters of the entities made with the common-input-ownership heuristic using OTC addresses found with heuristic 2.4

```
1  def combine_entities_with_otc_address(entities_collection,
        otc_collection):
2      entities = entities_collection.find({})
3      entities = [entity for entity in entities]
4      for entity in tqdm(entities):
5          for tx_hash in entity["tx_hashes"]:
6              otc = otc_collection.find_one({"$and": [{"tx_hash":
                    tx_hash}, {"heuristics.4": True}]})
7              if otc and otc["otc_output"]["address"][0] not in
                    entity["address_cluster"]:
8                  entity_with_otc_address =
                        entities_collection.find_one({"address_cluster":
                        otc["otc_output"]["address"][0]})
9                  if entity_with_otc_address and
                        entity_with_otc_address["_id"] != entity["_id"]:
10                     entities_collection.update_one({
11                         '_id': entity['_id']
12                     }, {
13                         '$addToSet': {
14                             'address_cluster': { '$each':
                                    entity_with_otc_address[
                                    "address_cluster"]}
15                         },
16                         '$push': {
17                             'tx_hashes': { '$each':
                                    entity_with_otc_address["tx_hashes"]}
18                         }
19                     })
20                     entities_collection.delete_one({"_id":
                            entity_with_otc_address["_id"]})
```

4.6 Testing

Unit testing was conducted on several algorithms using Python *unittest* package. Testing was implemented late in the development process. However, it did, in fact, expose flaws in the original implementation that had to be fixed. A small part of the testing script is shown in Algorithm 4.16, where Algorithm 4.10 for checking if a transaction has a self-change address is tested. The complete testing script is shown in Appendix C.5.

Algorithm 4.16: Unit testing on Algorithm 4.10

```
1      def test_has_self_change_address(self):
2          transaction = {
3              "inputs": [ { "address" : [ "address1" ] }, { "address" :
                   [ "address2" ] } ],
4              "outputs": [ { "address" : [ "address3" ] }, { "address" :
                   [ "address1" ] } ]
5          }
6          self.assertTrue(has_self_change_address(transaction))
7
8          transaction = {
9              "inputs": [ { "address" : [ "address1" ] }, { "address" :
                   [ "address2" ] } ],
10             "outputs": [ { "address" : [ "address3" ] }, { "address" :
                   [ "address4" ] } ]
11         }
12         self.assertFalse(has_self_change_address(transaction))
```

Chapter 5
Results

The results from running the implementations covered in chapter 4 show the different address reduction ratios achieved by the entities created by the *common-input-ownership heuristics* discussed in section 3.2.1, the number of OTC addresses determined by the heuristics discussed in section 3.2.2, and the reduction ratio achieved by combining them as discussed in section 3.2.3.

In this chapter, the heuristics are referred to in the following manner:

Common-input-ownership heuristics

- **H1.1** Heuristic 1.1 (Nakamoto, Satoshi) [1]
- **H1.2** Heuristic 1.2 (Ermilov *et al.*) [52]
- **H1.3** Heuristic 1.3 (our proposed heuristic)

One-time change address heuristics

- **H2.1** Heuristic 2.1 (Meiklejohn *et al.*) [31]
- **H2.2** Heuristic 2.2 (Zhang *et al.*) [13]
- **H2.3** Heuristic 2.3 (Ermilov *et al.*)] [52]
- **H2.4** Heuristic 2.4 (our proposed heuristic)

The reduction ratio is calculated by formula 5.1.

$$address_reduction = \frac{\#original_addresses - \#entities}{\#original_addresses} \tag{5.1}$$

#original_addresses is the combined number of unique input and output addresses, and *#entities* is the number of entities created by the clustering heuristic. Each output address not re-used as an input address is considered an entity unless H2.4 is involved in linking the address to an entity.

Generally, a higher reduction ratio value is better. However, the goal of combining our proposed heuristics ($H1.3 \times H2.4$) is to reach a reduction ratio similar to the one achieved by H1.1 - the clustering heuristic used by Graphsense. As discussed in chapter 3, this approach has a lower occurrence of false positives. There is no way to measure the ratio of false positives, but the properties of $H1.3 \times H2.4$ should ensure a lower occurrence compared to H1.1.

Because of time constraints and our limited resource access to the Graphsense API, the execution of all the heuristics for block $\sim 200,000$ did not completely finish, so this data has been left out of the results.

5.1 Common-input-ownership heuristics

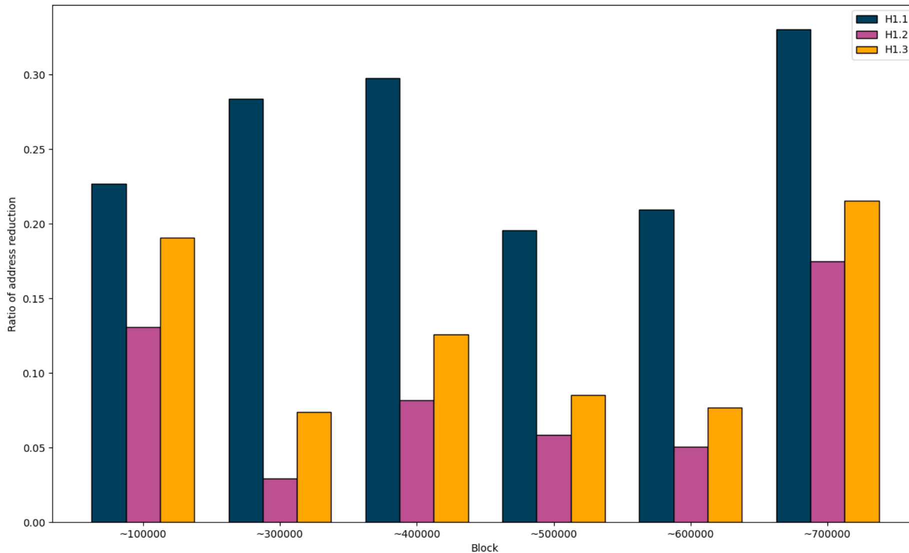


Figure 5.1: Reduction ratio of the common-input-ownership heuristics on data set 1

Figure 5.1 shows the different reduction ratios achieved by the *common-input-ownership heuristics*. It is clear that H1.1 performs best here when only taking the address reduction ratio into consideration. However, unlike H1.1, H1.2 should have no false positive results, and H1.3 should have no false positive results if we assume that its OTC heuristic (H2.4) does not include any false negative OTC addresses. In block 100,000, all the transactions have either one or two outputs, and the performance of H1.2 and H1.3 is almost on par with that of H1.1. Also, in the newest set of blocks, $\sim 700,000$, the performance is very good for both H1.2 and H1.3 compared to H1.1. However, because H1.2 performs worse than H1.3 in all cases, H1.2 is not included in

any more results. Additionally, H1.3 has reliable information about OTC addresses that can be used to reduce the number of entities by combining them as they are reused as inputs. More about this in section 5.3.

5.2 One-time change heuristics

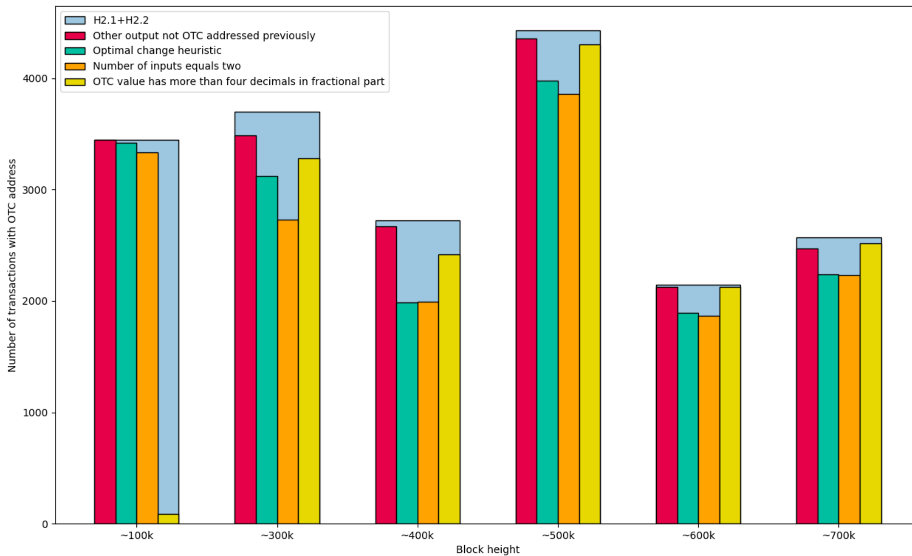


Figure 5.2: Impact of OTC properties on data set 1

Figure 5.2 shows the impact of different properties that can be used to reduce the number of false positive results in an OTC heuristic. In this plot, the number of OTC outputs found by H2.1+H2.2 is used as a baseline. This is part of the data that was analyzed when creating our proposed heuristic (H2.4) for finding the OTC address. As can be seen in the plot, limiting the valid transactions to those with two inputs and using the optimal change heuristic are the most impactful properties in most cases. The former was dropped from our proposed heuristic because it was considered too broad of an assumption, and it would exclude many multi-input transactions from H1.3. Assuming that an OTC output is only valid if the value has more than four decimals after the dot was also determined to have too big of an impact without providing enough value, this one was also excluded from H2.4. This can especially be seen for block $\sim 100,000$, where almost all OTC outputs are deemed invalid based on this assumption. This leaves us with the two properties used on H2.1+H2.2 for H2.4, which were both considered to reduce the number of false positives without excluding too many cases from the heuristic.

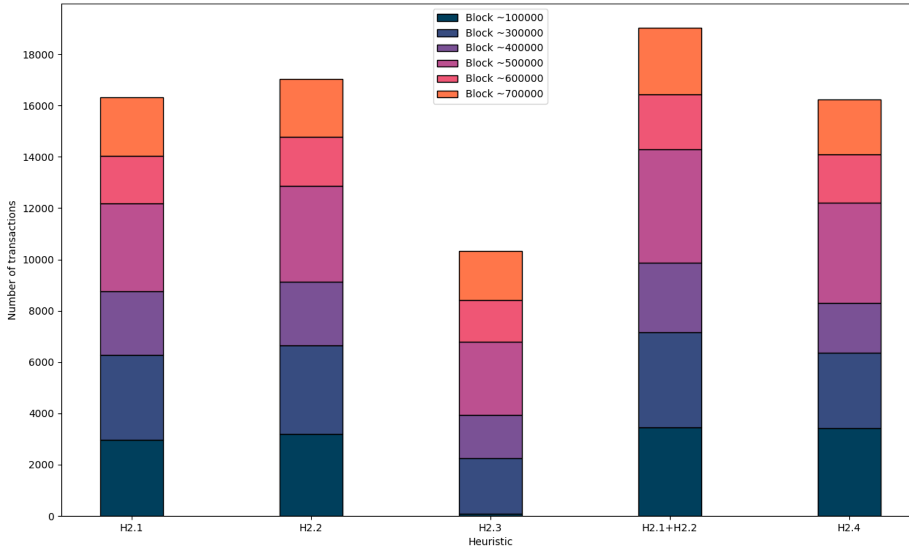


Figure 5.3: Volume of transactions with OTC address found using different heuristics on data set 1. Total number of transactions: 63,783

Figure 5.3 shows the volume of OTC addresses found using the heuristics discussed in section 3.2.2 on *data set 1*. H2.1+H2.2 has the highest number of false positives and performs the best, but while H1.3 performs the worst, it does have the lowest number of false positives. However, the number of false negatives is also the highest for this heuristic - leading to the bad performance of the heuristic. Our proposed heuristic (H2.4) performs as well as H2.1 and a little worse than H2.1+H2.2, but with a definite lower number of false positives. Because of this, we consider H2.4 to be the best heuristic for finding the OTC address.

5.3 Combination of common-input-ownership heuristic and OTC heuristic

As explained in section 3.2.3, the OTC heuristics not only link output addresses to the sender, but can also be used to combine clusters when the address is reused as an input address. By adding H2.4 to H1.3 ($H2.4 \times H1.3$), similar, and in some cases, even better reduction ratios are achieved than by H1.1. This is shown on *data set 1* in Table 5.1, and visualized in Figure 5.4.

For block $\sim 100,000$, $H1.3 \times H2.4$ has a 100% improved reduction ratio compared to H1.1. There were fewer Bitcoin users for these transactions compared to the other parts of the data sets, as shown by its number of unique addresses. The $\sim 10,000$

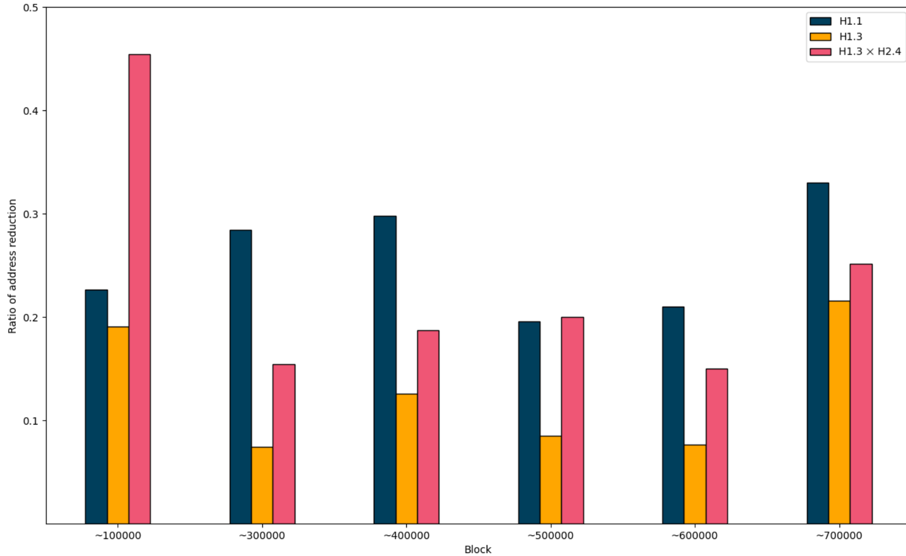


Figure 5.4: Performance of address clustering heuristics on data set 1

transactions for Block 100,000 were performed over a longer period of time with a higher prevalence of change address reuse, leading to a high occurrence of entity combining. Additionally, no coin mixing services existed at the time. For the other parts of the data set, the performance of $H1.3 \times H2.4$ compared with H1.1 varies. However, the results are all favorable because of the emphasis on excluding false positives in the clustering.

Table 5.1: Results of address clustering heuristics on data set 1

Block	Unique addresses	Entities			Percent reduction		
		H1.1	H1.3	H1.3 × H2.4	H1.1	H.3	H1.3 × H2.4
~100,000	12,644	9,777	10,230	6,899	22,67	19,09	45,44
~300,000	36,212	25,927	33,534	30,627	28,40	7,40	15,42
~400,000	31,536	22,149	27,571	25,625	29,77	12,57	18,74
~500,000	34,008	27,354	31,117	27,216	19,57	8,50	19,97
~600,000	25,646	20,269	23,678	21,805	20,97	7,67	14,98
~700,000	59,160	39,626	46,411	44,284	33,02	21,55	25,15

5.3.1 Reduction over time

To see if similar results for $H1.3 \times H2.4$ on Block $\sim 100,000$ could be achieved for newer blocks, the implementations were run on data set 2 with a higher number

of consecutive transactions. Optimally, this data set should have included millions of transactions conducted over several weeks, but that was not possible given our limited resources and time frame. *Data set 2* has all the transactions for one specific date - 144 blocks containing 721,253 transactions conducted on Feb. 1, 2022. Block $\sim 100,000$ for *data set 1* has 2348 blocks containing 10,072 transactions conducted over 15 days. Nevertheless, the goal was to find a trend in how the reduction ratio changed for the different heuristics as more transactions were involved in the clustering. The data set was divided into several subsets of (10000, 20000, ... 260000, 270000, 271253) transactions, where the heuristics were executed individually. The results can be seen in Table 5.2, and is visualized in Figure 5.5.

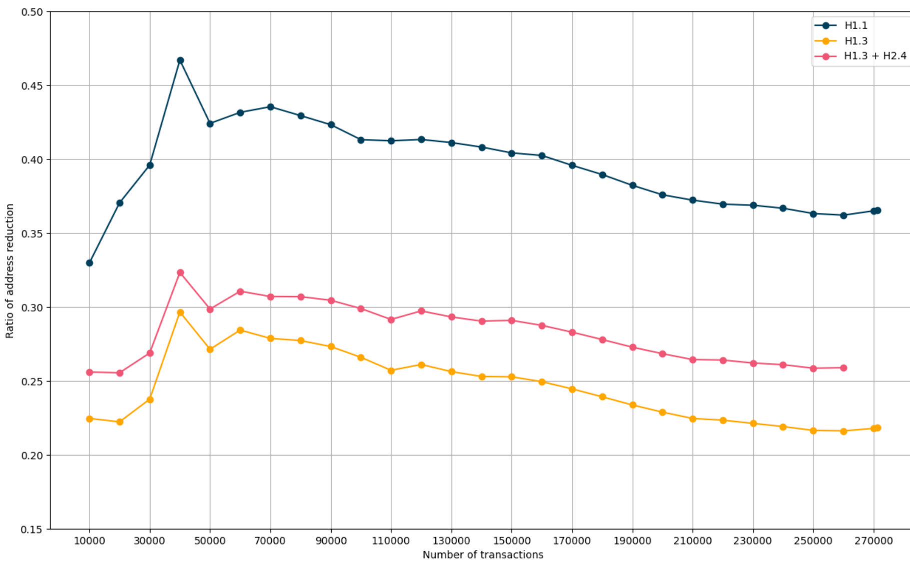


Figure 5.5: Performance of address clustering heuristics on a growing volume of transactions on data set 2

As shown in Figure 5.5, the heuristics follow almost the exact same pattern as more transactions are analyzed. The performance of $H1.3 \times H2.4$ compared to H1.1 varies by a few percent with an average of 71.30%. Since these are the most recent and relevant transactions for address clustering, we use this number as the conclusive result for $H1.3 \times H2.4$ compared to H1.1. However, a bigger data set could achieve more significant results. More about this in section 6.2.

Table 5.2: Results of address clustering heuristics on a growing volume of transactions on data set 2

		Entities			Percent reduction		
Transactions	Unique addresses	H1.1	H1.3	H1.3 × H2.4	H1.1	H1.3	H1.3 × H2.4
10,000	56,687	37,990	43,952	42,174	32,98	22,47	25,60
20,000	112,261	70,674	87,301	83,575	37,04	22,23	25,55
30,000	171,908	103,830	131,051	125,680	39,60	23,77	26,89
40,000	260,852	139,029	183,474	176,495	46,70	29,66	32,34
50,000	322,214	185,528	234,751	225,991	42,42	27,14	29,86
60,000	391,632	222,581	280,249	269,958	43,17	28,44	31,07
70,000	440,253	248,541	317,508	305,049	43,55	27,88	30,71
80,000	488,815	278,888	353,258	338,753	42,95	27,73	30,70
90,000	534,208	308,062	388,214	371,500	42,33	27,33	30,46
100,000	572,302	335,814	420,055	401,189	41,32	26,60	29,90
110,000	618,732	363,562	459,597	438,315	41,24	25,72	29,16
120,000	665,536	390,456	491,747	467,640	41,33	26,11	29,73
130,000	714,702	420,809	531,503	505,088	41,12	25,63	29,33
140,000	769,384	455,323	574,698	545,901	40,82	25,30	29,05
150,000	813,986	484,936	608,207	577,136	40,42	25,28	29,10
160,000	871,795	520,889	654,264	621,087	40,25	24,95	28,76
170,000	914,244	552,345	690,564	655,490	39,58	24,47	28,30
180,000	954,890	582,845	726,380	689,411	38,96	23,93	27,80
190,000	994,868	614,514	762,297	723,419	38,23	23,38	27,28
200,000	1,034,265	645,478	797,510	756,622	37,59	22,89	26,84
210,000	1,073,340	673,758	832,256	789,478	37,23	22,46	26,45
220,000	1,110,999	700,404	862,695	817,543	36,96	22,35	26,41
230,000	1,155,028	729,017	899,420	852,240	36,88	22,13	26,21
240,000	1,195,073	756,727	933,170	883,166	36,68	21,92	26,10
250,000	1,234,012	785,755	966,759	914,864	36,33	21,66	25,86
260,000	1,273,598	812,382	998,201	585,647	36,21	21,62	25,90
270,000	1,324,651	841,130	1,035,998	605,966	36,50	21,79	
271,146	1,330,897	844,353	1,040,132	607,953	36,56	21,85	

Chapter 6

Future work

6.1 Difficulties in detecting centralized coin mixing

We have explored various possibilities in detecting CoinJoin transactions in section 3.3.3. Identifying such transactions is trivial because they always include multiple inputs/outputs. Suppose they include sub-transactions as described in section 3.3.3. In that case, it is almost certainly a CoinJoin transaction, and a common ownership behind the addresses in the sub-transaction [60]. However, the problem becomes increasingly more difficult if other types of *coin mixing* protocols are considered. Despite the weaknesses of centralized *coin mixers* described in section 3.3.2, detecting such transactions is difficult, due to the similarities they share with regular transactions [55]. As mentioned in section 3.3.2, centralized mixers act as intermediaries between any two parties in a transaction. They can receive and forward transactions with any number of inputs/outputs [55].

Figure 6.1 illustrates how users can mask their ownership of the Bitcoin value by using a centralized mixing service. User 1 sends 1.0 BTC to an arbitrary *coin mixing* server. This server later returns 0.99 BTC to the sender from the output address of User 2's transaction of 1.99 BTC. The 0.99 BTC returned to User 1 looks like it originated from User 2 when the Bitcoin values were only obfuscated in a way that made them appear on different addresses.

The main problem with mixing transactions in the way described in Figure 6.1 is that they are hard to detect by simply looking at the public blockchain. A more reliable way to detect them involves tracing network traffic as described in section 1.4.2. Furthermore, mixing servers usually have a large volume of incoming and outgoing network traffic. Therefore, identifying their IP addresses should be a feasible task [7]. For example, suppose the IP addresses of mixing servers can be identified. In that case, this server's incoming and outgoing transactions are labeled as mixed transactions. This can be used with other deanonymization methods, such as matching transactions having this label with other transactions of similar values.

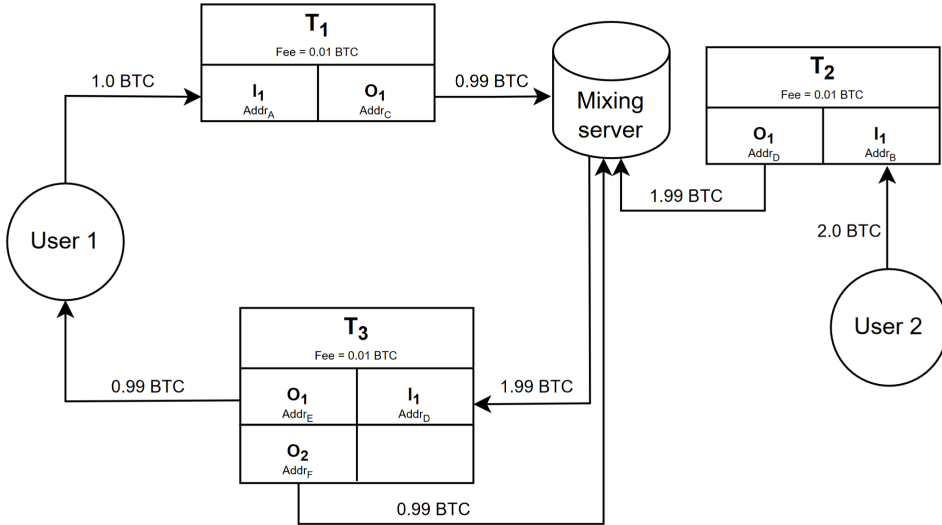


Figure 6.1: Example of a mixing transaction through a centralized mixing server

6.2 Analyzing a larger quantity of transactions

In this thesis, Graphsense is used for fetching transactions and other relevant data for performing address clustering. Since the number of transactions used for analyses was limited to 73,798 for *data set 1* and 271,146 for *data set 2*, the number of possible reductions were restricted. The number of transactions did not provide enough data for exploiting all the privacy weaknesses promoted by address reuse. Using a data set containing millions of transactions conducted over several weeks could show a more significant difference in the heuristic's performance over time, possibly making an even stronger case for the validity of our proposed heuristics. This could have been achieved by running the Graphsense demo locally or getting access to perform an unlimited amount of requests per hour to the remote Graphsense API we were provided. Additionally, as mentioned in section 4.4.3, Graphsense's *Addresses_api.list_address_txs* function's lack of parameters for filtering an address's transaction based on if it had been used as an output address or input address caused a bottleneck when executing our implementation. Improving this functionality would have severely reduced the number of required requests and the runtime. An alternative way of accessing the blockchain's data is to run our own Bitcoin client. The client contains the entire blockchain, with a size of roughly 410 GB [68]. However, by doing this, we would have had to implement our own variations of the functions provided by Graphsense that we utilized for fetching relevant data for transactions, blocks, and addresses.

6.3 Combining clustering with direct deanonymization techniques

We have proposed a few methods of address clustering in this thesis. It is possible to achieve significant reductions on the set of possible entities by studying part of the Bitcoin blockchain alone, with the help of Graphsense. Other deanonymization methods are mentioned briefly in section 1.4. These can be used in conjunction with address clustering techniques.

A considerable problem with studying many transactions at once is that many of them will be insignificant and not help us discover criminal activities. Achieving reductions on possible entities based on seemingly ordinary user transactions grants us nothing other than compromising the privacy of innocent users. The K-means algorithm discussed in section 1.4.3 can be used in this context to differentiate between clusters in the Bitcoin blockchain. Suppose we can differentiate between "good" and "bad" clusters of transactions before analyzing addresses. In that case, much time can be saved by not analyzing transactions that are less important.

We can also set up a Bitcoin core node of our own and participate in the Bitcoin network as a passive eavesdropper [7]. By analyzing the network packets transmitted by the participants in the network, it is possible to link Bitcoin addresses to IP addresses. However, this is only possible with a maintained connection to each participant in the network. For example, it is impossible to trace the IP addresses of transactions that have been performed in the past. Nonetheless, it is a feasible method for potentially linking more addresses to an identity outside the scope of address clustering.

Transaction graph analysis is mentioned in section 1.4.1 as a viable method of tracing flows that have been made between certain participants in the Bitcoin network. Such graphs, however, tend to get very large, making analysis difficult. However, address clustering heuristics helps tremendously with graph analysis because several vertices can be grouped into one. Furthermore, vertex grouping can also be done with network analysis, as mentioned in the previous paragraph.

6.4 The CoinJoin sub-transaction problem

Algorithm 3.1 proposed in section 3.3.3 can be used for analyzing the input and output values of a CoinJoin transaction, and find sub-transactions within. In practice, it can be included alongside the common ownership and OTC heuristics proposed in section 3.2, to perform clustering on transactions with several inputs/outputs. However, since the algorithm is NP-complete, it can delay the clustering processes significantly. The asymptotic run time of Algorithm 3.1 is $O(2^N)$, which means that

the run time will double for each new element. It is likely better to run this algorithm exclusively on a selection of transactions instead.

Since roughly a quarter of all illicit payments in Bitcoin are done with the use of *coin mixing* [55], it might be better to execute this algorithm conservatively, i.e., executing the algorithm exclusively on large transactions or those suspected to be mixed with CoinJoin. Our analysis can be restricted to be within transactions that are of a specific input/output size or higher (Figure 3.3a and 3.5 tell us that these transactions do not make up a large quantity, and as a result running them should not be slow).

A similar issue was explored by Atlas in 2014 [69]. They created a tool written in php called *CoinJoin Sudoku*, which analyses sums within CoinJoin transactions, similar to what Algorithm 3.1 does. Due to the inefficiencies and slowness of their tool, they could only test it on a few selected transactions on the blockchain. They were able to group roughly 69% of the inputs with 53% of the output addresses, and they established a 100% common ownership between the found addresses [69]. This result is good because there are likely no false positives; however, quite a few remaining inputs did not get matched in groups. The *transaction fees* likely skew the output values, making them harder to be matched together in groups.

Transaction fees are not considered in Algorithm 3.1. In the CoinShuffle protocol [58] proposed by Ruffing, Sanchez, and Kate, it is suggested that the *transaction fee* μ is spent equally among the participants. Each output value should therefore be reduced by μ/N, where N is the number of participants. Let us assume that the *transaction fee* is spent equally in every CoinJoin transaction. In that case, it is straightforward to consider *transaction fees* when the output values can be adjusted relative to the *transaction fee*. However, this is not realistic in practice because, in a CoinJoin transaction, each participant agrees on the output values before the transaction occurs [8]. The *transaction fee* can therefore be unevenly distributed among the participants, and is almost impossible to differentiate.

Here are a few proposed ideas on how to handle *transaction fees* in a CoinJoin transaction:

If the transaction fee is small This is the simplest case to handle because the *transaction fee* is likely small compared to all input and output values. Therefore, the *transaction fee* can be a margin of error when calculating the subset sums of the output values. This could result in false positives, but with a small likelihood due to the low value of the *transaction fee*.

Finding sub-transactions first In the case of Atlas [69], sub-transactions are found without considering the *transaction fees*. This means that it is still

possible to find sub-transactions, likely because there are output values not reduced by the *transaction fee*. Once there are no more groups to find, one can start using the *transaction fee* as a margin of error. This can generate a lot of false positives if the *transaction fee* is large.

There is also one edge-case transaction on which it is impossible to perform address clustering, namely *perfect CoinJoin* transactions [69]. Figure 6.2 shows an example of a perfect CoinJoin transaction, where three participants transact 1.01 BTC to other participants with a *transaction fee* of 0.03 BTC evenly split evenly among the output addresses. Because all the inputs are identical, and the outputs are identical, the amounts reveal no information on which 1.01 BTC inputs correspond to which 1 BTC outputs. Therefore, the chance of co-ownership of each address is one out of three [69]. Such transactions are sporadic but could be very dangerous if performed with large amounts and many participants.

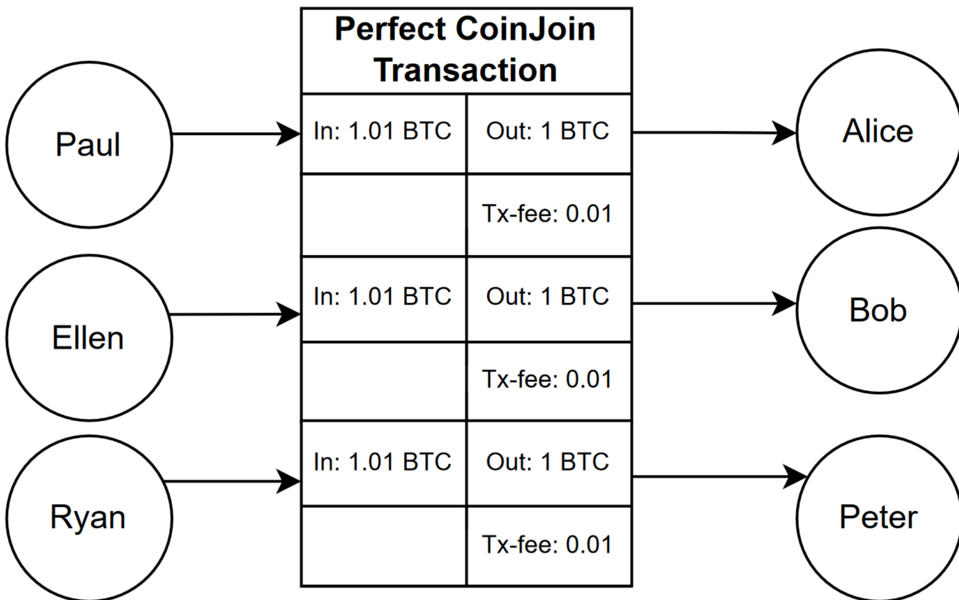


Figure 6.2: Example of a perfect CoinJoin transaction

Chapter 7
Conclusion

Bitcoin is by many considered to be an anonymous form of digital payment. However, its degree of anonymity is not as prevalent as it may seem. The pseudo-anonymous nature of Bitcoin transactions which are publicly available on the blockchain enables the use of deanonymization techniques that can be used to reveal the identities of users. Regardless, there has been increased activity in using cryptocurrencies for illicit transactions. Therefore, we analyzed deanonymization techniques that can be beneficial for law-enforcement agencies to investigate criminal activity involving cryptocurrency transactions.

This thesis focuses on deanonymization techniques for Bitcoin transactions based on address clustering, implemented using the Graphsense Cryptoasset Analytics platform. This platform's built-in address clustering method has significant weaknesses that we have analyzed and improved. Our proposed clustering heuristics are based on excluding transactions obfuscated by *coin mixing* and including clustering properties enabled by reliably finding one-time change addresses for transactions. By using the heuristics proposed in section 3.2 and comparing their performance to existing ones, we have achieved similar reductions on the address sets, but with less false positive results. We have also shown that many of the transactions obfuscated by *coin mixing* that we have chosen to exclude in our clustering heuristics can be analyzed and dissected to be used for clustering. Based on our research, we recommend Graphsense consider changing their clustering technique to our variation, which is more robust and reliable.

In the context of this thesis work, *coin mixing* strategies are still a considerable concern. During our research, a few weaknesses were discovered with the CoinJoin protocol in particular, which can be exploited to achieve address clustering results. We proposed a few methods of doing this in practice. However, due to the complexity of this problem, we did not perform any tests on real transactions. This should be explored in more detail in future work.

References

[1] Satoshi Nakamoto. «Bitcoin: A peer-to-peer electronic cash system». In: *Decentralized Business Review* (2008), p. 21260.

[2] Elli Androulaki et al. «Evaluating User Privacy in Bitcoin». In: *Financial Cryptography and Data Security*. Ed. by Ahmad-Reza Sadeghi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 34–51. ISBN: 978-3-642-39884-1.

[3] Alex Biryukov and Sergei Tikhomirov. «Deanonymization and Linkability of Cryptocurrency Transactions Based on Network Analysis». In: *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. 2019, pp. 172–184. DOI: 10.1109/EuroSP.2019.00022.

[4] «THE 2020 STATE OF CRYPTO CRIME - Everything you need to know about darknet markets, exchange hacks, money laundering and more». In: (2020). URL: https://tinyurl.com/2020cryptocrime.

[5] Harald Vranken. «Sustainability of bitcoin and blockchains». In: *Current Opinion in Environmental Sustainability* 28 (2017). Sustainability governance, pp. 1–9. ISSN: 1877-3435. DOI: https://doi.org/10.1016/j.cosust.2017.04.011. URL: https://www.sciencedirect.com/science/article/pii/S1877343517300015.

[6] Primavera De Filippi and Benjamin Loveluck. «The invisible politics of bitcoin: governance crisis of a decentralized infrastructure». In: *Internet policy review* 5.4 (2016).

[7] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. «Deanonymisation of clients in Bitcoin P2P network». In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, pp. 15–29.

[8] Gregory Maxwell. *CoinJoin: Bitcoin privacy for the real world*. en. URL: https://bitcointalk.org/index.php?topic=279249.0 (last visited: May 12, 2021).

[9] *Mastering Bitcoin*. original-date: 2013-08-11T23:18:28Z. June 2022. URL: https://github.com/bitcoinbook/bitcoinbook/blob/77b91b1949e2c03a36c395586a44dac20ec41533/ch06.asciidoc (last visited: June 7, 2022).

[10] Fangfang Dai et al. «From Bitcoin to cybersecurity: A comparative study of blockchain application and security issues». In: *2017 4th International Conference on Systems and Informatics (ICSAI)*. 2017, pp. 975–979. DOI: 10.1109/ICSAI.2017.8248427.

[11] QingChun ShenTu and Jianping Yu. «Research on Anonymization and De-anonymization in the Bitcoin System». In: *CoRR* abs/1510.07782 (2015). arXiv: 1510.07782. URL: http://arxiv.org/abs/1510.07782.

[12] Bernhard Haslhofer, Roman Karl, and Erwin Filtz. «O Bitcoin Where Art Thou? Insight into Large-Scale Transaction Graphs». In: *SEMANTiCS*. 2016.

[13] Yuhang Zhang, Jun Wang, and Jie Luo. «Heuristic-Based Address Clustering in Bitcoin». In: *IEEE Access* 8 (2020), pp. 210582–210591. DOI: 10.1109/ACCESS.2020.3039570.

[14] Jason Hirshman, Yifei Huang, and Stephen Macke. «Unsupervised approaches to detecting anomalous behavior in the bitcoin transaction network». In: *3rd ed. Technical report, Stanford University* (2013).

[15] Zero-Knowledge Proof: How it Works Applications in 2022. URL: https://research.aimultiple.com/zero-knowledge-proofs/ (last visited: May 2, 2022).

[16] Federal Communications Commission. *What are Coin Mixers, and how do they work?* en. URL: https://www.worldcryptoindex.com/what-is-a-coin-mixer/ (last visited: May 18, 2021).

[17] «Request for Proposal 2032H8-20-R-00500». In: (Sept. 2020). URL: https://sam.gov/api/prod/opps/v3/opportunities/resources/files/bb0247a3accc46beb2901af74b78438d/download?&status=archived&token=.

[18] *Chainalysis*. en. URL: https://www.forbes.com/companies/chainalysis/ (last visited: Nov. 11, 2021).

[19] Anil Gaihre, Santosh Pandey, and Hang Liu. «Deanonymizing cryptocurrency with graph learning: the promises and challenges». In: *2019 IEEE Conference on Communications and Network Security (CNS)*. IEEE. 2019, pp. 1–3.

[20] Fergal Reid and Martin Harrigan. «An analysis of anonymity in the bitcoin system». In: *Security and privacy in social networks*. Springer, 2013, pp. 197–223.

[21] Giulia Fanti and Pramod Viswanath. «Anonymity properties of the bitcoin P2P network». In: *arXiv preprint arXiv:1703.08761* (2017).

[22] Alex Biryukov and Ivan Pustogarov. «Bitcoin over Tor isn't a good idea». In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 122–134.

[23] Patrick Monamo, Vukosi Marivate, and Bheki Twala. «Unsupervised learning for robust Bitcoin fraud detection». In: *2016 Information Security for South Africa (ISSA)*. IEEE. 2016, pp. 129–134.

[24] Kristina P Sinaga and Miin-Shen Yang. «Unsupervised K-means clustering algorithm». In: *IEEE access* 8 (2020), pp. 80716–80727.

[25] *Documentation – AIT Graphsense*. URL: https://graphsense.info/documentation/ (last visited: June 2, 2022).

[26] Bernhard Haslhofer et al. «GraphSense: A General-Purpose Cryptoasset Analytics Platform». In: *Arxiv pre-print* (2021). URL: https://arxiv.org/abs/2102.13613.

[27] *NEWS / PRESS | TITANIUM*. URL: https://titanium-project.eu/faq/index.html (last visited: Nov. 10, 2021).

[28] *TITANIUM: Tools for the Investigation of Transactions in Underground Markets.* URL: https://titanium-project.eu/ (last visited: Nov. 10, 2021).

[29] *About Graphsense – AIT Graphsense.* URL: https://graphsense.info/about/ (last visited: June 9, 2022).

[30] *AIT Graphsense – AIT Austrian Institute of Technology.* URL: https://graphsense.info/ (last visited: June 5, 2022).

[31] Sarah Meiklejohn et al. «A Fistful of Bitcoins: Characterizing Payments among Men with No Names». In: *Commun. ACM* 59.4 (Mar. 2016), pp. 86–93. ISSN: 0001-0782. DOI: 10.1145/2896384. URL: https://doi.org/10.1145/2896384.

[32] *Contact – AIT Graphsense.* URL: https://graphsense.info/contact/ (last visited: June 9, 2022).

[33] *Bitcoin Transactions Per Day.* URL: https://ycharts.com/indicators/bitcoin_transactions_per_day (last visited: June 5, 2022).

[34] *Openstack at NTNU - SkyHigh - NTNU Wiki.* URL: https://www.ntnu.no/wiki/display/skyhigh (last visited: June 2, 2022).

[35] *What is MongoDB - Working and Features.* en-us. Section: Advanced Computer Subject. Jan. 2020. URL: https://www.geeksforgeeks.org/what-is-mongodb-working-and-features/ (last visited: June 2, 2022).

[36] *JSON And BSON.* en-us. URL: https://www.mongodb.com/json-and-bson (last visited: June 2, 2022).

[37] *Robo 3T - Javatpoint.* en. URL: https://www.javatpoint.com/robo-3t (last visited: June 2, 2022).

[38] *Python 3.9 Documentation.* URL: https://docs.python.org/3.9/ (last visited: June 5, 2022).

[39] *Documentation for Visual Studio Code.* en. URL: https://code.visualstudio.com/docs (last visited: June 5, 2022).

[40] The pip developers. *pip: The PyPA recommended tool for installing Python packages.* URL: https://pip.pypa.io/ (last visited: June 5, 2022).

[41] *graphsense-python.* original-date: 2020-06-24T14:41:16Z. May 2022. URL: https://github.com/graphsense/graphsense-python (last visited: June 5, 2022).

[42] *Matplotlib — Visualization with Python.* URL: https://matplotlib.org/ (last visited: June 6, 2022).

[43] *PyMongo 4.1.1 Documentation — PyMongo 4.1.1 documentation.* URL: https://pymongo.readthedocs.io/en/stable/ (last visited: June 6, 2022).

[44] *python-dotenv · PyPI.* URL: https://pypi.org/project/python-dotenv/ (last visited: June 11, 2022).

[45] *Git.* URL: https://git-scm.com/ (last visited: June 6, 2022).

[46] *Blockchain.com Explorer | BTC | ETH | BCH.* en. URL: https://www.blockchain.com/explorer (last visited: June 6, 2022).

[47] Ujan Mukhopadhyay et al. «A brief survey of Cryptocurrency systems». In: *2016 14th Annual Conference on Privacy, Security and Trust (PST)*. 2016, pp. 745–752. DOI: 10.1109/PST.2016.7906988.

[48] *What Happened to Bitcoin's Transaction Volume?* en-US. URL: https://finance.yahoo.com/news/happened-bitcoins-transaction-volume-172321434.html (last visited: June 6, 2022).

[49] Rostislav Skudnov. «Bitcoin clients». In: (2012).

[50] *P2P Network — Bitcoin*. URL: https://developer.bitcoin.org/devguide/p2p_network.html (last visited: June 7, 2022).

[51] Felix Konstantin Maurer, Till Neudecker, and Martin Florian. «Anonymous CoinJoin Transactions with Arbitrary Values». In: *2017 IEEE Trustcom/BigDataSE/ICESS*. 2017, pp. 522–529. DOI: 10.1109/Trustcom/BigDataSE/ICESS.2017.280.

[52] Dmitry Ermilov, Maxim Panov, and Yury Yanovich. «Automatic Bitcoin Address Clustering». In: *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. Dec. 2017, pp. 461–466. DOI: 10.1109/ICMLA.2017.0-118.

[53] Jonas David Nick. «Data-Driven De-Anonymization in Bitcoin». en. Master's Thesis, Distributed Computing Group Computer Engineering and Networks Laboratory, ETH Zürich, August 9, 2015. MA thesis. Zürich: ETH Zurich, 2015. DOI: 10.3929/ethz-a-010541254.

[54] How Popular Are Crypto Mixers? Here's What the Data Tells Us. URL: https://www.coindesk.com/layer2/privacyweek/2022/01/25/how-popular-are-crypto-mixers-heres-what-the-data-tells-us/ (last visited: June 10, 2022).

[55] What is a cryptocurrency mixer and how does it work? URL: https://cointelegraph.com/explained/what-is-a-cryptocurrency-mixer-and-how-does-it-work (last visited: June 10, 2022).

[56] Bitcoin blender. URL: https://blendar.io/ (last visited: June 10, 2022).

[57] Sygna. *Coin Mixers Draw Regulatory Heat in US and UK Due to AML Concerns*. en. URL: https://www.sygna.io/blog/coin-mixer-regulation-in-us-and-uk/ (last visited: May 18, 2021).

[58] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. «Coinshuffle: Practical decentralized coin mixing for bitcoin». In: *European Symposium on Research in Computer Security*. Springer. 2014, pp. 345–364.

[59] 9 Best Bitcoin Mixers and Tumblers in 2021. URL: https://beincrypto.com/learn/best-bitcoin-mixers/ (last visited: May 18, 2022).

[60] Artem A Maksutov et al. «Detection of blockchain transactions used in blockchain mixer of coin join type». In: *2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*. IEEE. 2019, pp. 274–277.

[61] How Does This Bitcoin Privacy Improvement Compare with CoinJoin and CoinShuffle? URL: https://coinjournal.net/news/bitcoin-privacy-improvement-compare-coinjoin-coinshuffle/ (last visited: May 21, 2022).

[62] Xanda Schoefield and Eva Tardos. «Subset Sum is NP-complete». en. In: (), p. 3. URL: https://www.cs.cornell.edu/courses/cs4820/2018fa/lectures/subset_sum.pdf.

[63] Print sums of all subsets of a given set. URL: https://www.geeksforgeeks.org/print-sums-subsets-given-set/ (last visited: June 8, 2022).

[64] *Terms*. en. URL: https://graphsense.info/terms (last visited: June 10, 2022).

[65] *nohup(1) - Linux man page*. URL: https://linux.die.net/man/1/nohup (last visited: June 10, 2022).

[66] Thomas N. Hibbard. «Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting». In: *J. ACM* 9.1 (Jan. 1962), pp. 13–28. ISSN: 0004-5411. DOI: 10.1145/321105.321108. URL: https://doi.org/10.1145/321105.321108.

[67] *Multikey Indexes — MongoDB Manual*. en. URL: https://www.mongodb.com/docs/manual/core/index-multikey/ (last visited: June 12, 2022).

[68] Bitcoin Blockchain size. URL: https://ycharts.com/indicators/bitcoin_blockchain_size (last visited: June 16, 2022).

[69] Weak Privacy Guarantees for SharedCoin Mixing Service. URL: http://www.coinjoinsudoku.com/advisory/ (last visited: May 18, 2022).

Appendix A

Communication with the Graphsense team

GraphSense API key

noreply@graphsense.info <noreply@graphsense.info>
ma. 14.02.2022 08:54
Til: Håvard Hunshamar <havard.hunshamar@ntnu.no>

Thanks for your interest in GraphSense!

As requested, we are providing you an API key to our demo:

With that key you can
- use the GraphSense dashboard demo in your browser: demo.graphsense.info
- use our REST API demo (https://api.graphsense.info/ui/)

Before using our demo, please read our terms and conditions of use:
https://graphsense.info/terms.html

Please note that our server capacities are limited, which is the reason why we restricted your API Key to 1000 requests per hour.

If you have any questions, please consult the GraphSense Documentation and FAQ page (https://graphsense.info/documentation.html) or, if you don't find the answer, ask us directly (contact@graphsense.info).

We hope you enjoy GraphSense and also add you to our users list, through which we send updates and release announcements.

Regards,
The GraphSense Team

--
This is an auto-generated email - please do not reply directly to this message.

Re: Graphsense request limit

graphsense-contact <graphsense-contact@ait.ac.at>
ma. 30.05.2022 09:19
Til: Håvard Hunshamar <havard.hunshamar@ntnu.no>

Hello,

I've increased your rate limit to 10000, for the next three weeks.

Best regards,
Melitta Dragaschnig of the GraphSense Team

From: Håvard Hunshamar <havard.hunshamar@ntnu.no>
Sent: Wednesday, May 25, 2022 10:24
To: graphsense-contact
Subject: Graphsense request limit

Hello!

I am using graphsense for my master thesis, and I am wondering if there is any way I could get a higher rate limit for just a few weeks. I am delivering my thesis in 3 weeks, and it would be very helpful if I could do more than 1000 requests per hour just for 2-3 weeks.

My API key is :

Have a nice day!

--
Håvard Hunshamar

Appendix B

Request for resources in Openstack

Ønske om ressurser i Openstack

Beskriv ditt ressursbehov

Beskriv prosjektet kort	Masteroppgave i kommunikasonsteknologi, skal kjøre et eksisterende prosjekt som krever god maskinvare. https://graphsense.info/documentation.html
Tilhørighet	Forskning/undervisning IIK (SkyHiGh)
Trengs ressursene ifm arbeid i et spesifikt emne, oppgi emnekode	TTM4905
Estimert antall CPU-kjerner	16
Estimert mengde RAM (GB)	128
Estimert lagringsplass (GB)	256
Hvilken type ressurs trenger du?	Én ferdiginstallert maskin
Hvilket operativsystem?	ubuntu
Hvis du trenger Linux, legg ved den offentlige nøkkelen fra et SSH-nøkkelpar du ønsker å bruke for innlogging	ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQCyXoh6OP2Z8ljc9SWx nsUoPeYM0WeBi13aMcIpxfKisYSknmooA4T3R49bMbPum1kv1wj qF69ys51Haiy+rwK2FVFbOpWXeKtTuWQm1pYEFi+vAqSlok6n7w kVafj8FwsTIOcT+EixFG62S+4WZi/eq97cDXj/z0fGmqRnSwLwjH86 SMz37v+Wc1V2bG5DmPev0ktI9jDMH53Hvwl/NFfl3V3TT6ukL1w/q 5LfRch9oBACV1azAo7bWBbVQkvtNYCpgIHUI5UT/CnBJV9Js8Sa yw2hVxgDmN4ZuPD+q8QPmYOlsXiduHjrvuwQoCakMUsjW6TN3 q1waLDLDbO/+h5J0/8DCQUmCdlK1K4NRzthT+lhGNWkRbAOU1 SbDHSUirMRLiKmasjQOyqM+XXIafkK5AP3Cr13KJW7if/zYKI949 QSys6qniwR9OxSz8MIp2n0DNYo0EwKwhLAoMQ8wtK3j9J6fR6ffg Z4mbrZjecH+dLAAgFComzHs9pNtaY/7s8= haavarhu@stud.ntnu.no
Som standard åpnes det kun for SSH/RDP/ICMP. Trenger du fler åpne porter? Angi hvilke (protokoll/port):	
Hvor lenge trenger du ressursene?	1. august 2022 00:00
Tjenestebeskrivelsen er lest og forstått	✔

Appendix C
Implementations

C.1 Populating data sets

```
import os
import graphsense
from dotenv import load_dotenv
from graphsense.api import blocks_api, general_api, bulk_api
import datetime
from time import sleep
import pymongo
from tqdm import tqdm

load_dotenv('.env')

api_key = os.environ.get("api_key")
configuration =
    graphsense.Configuration(host="https://api.graphsense.info")
configuration.api_key["api_key"] = api_key
connectURI = os.environ["connectURI"]
client = pymongo.MongoClient(connectURI)

db = client["master"]

api_client = graphsense.ApiClient(configuration)
blocks_api = blocks_api.BlocksApi(api_client)
general_api = general_api.GeneralApi(api_client)
bulk_api = bulk_api.BulkApi(api_client)

def get_io(tx_hashes, io):
    io_list = []
    i = 0
    while i < len(tx_hashes):
        try:
            body = {
                "tx_hash": tx_hashes[i:i+50], "io": io}
            io_list.extend(bulk_api.bulk_json(
                'btc', 'get_tx_io', 1, body, async_req=True).get())
        except graphsense.ApiException as e:
```

```
35              # Request limit exceeded
36              if (e.status == 429):
37                  sleep(int(e.headers["Retry-After"]) + 60)
38                  continue
39              else:
40                  raise e
41          i += 50
42      return io_list
43
44  def format_input_output(puts):
45      formatted_puts = []
46      for put in puts:
47          if "_info" in put and put["_info"] == "no␣data":
48              continue
49          address = []
50          try:
51              address = [put["address"][0][""]]
52          except IndexError:
53              pass
54          formatted_puts.append({"_request_tx_hash":
                 put["_request_tx_hash"], "address": address, "value":
                 {"fiat_values" : [{"code": "eur", "value":
                 put["value_eur"]}, {"code": "usd", "value":
                 put["value_usd"]}], "value": int(put["value_value"])}})
55      return formatted_puts
56
57  def format_value(value):
58      return {"value": int(value["value"]), "fiat_values": [{"code":
           fv["code"], "value": fv["value"]} for fv in
           value["fiat_values"]]}
59
60  def format_transactions(transactions, inputs, outputs):
61      formatted_transactions = []
62      inputs = format_input_output(inputs)
63      outputs = format_input_output(outputs)
64      for transaction in transactions:
65          fortmatted_transaction = {"_id": transaction["tx_hash"],
                 "coinbase": transaction["coinbase"], "height": None,
                 "inputs": [], "outputs": [], "timestamp":
                 transaction["timestamp"], "total_input": {},
                 "total_output": {}, "tx_hash": transaction["tx_hash"],
                 "tx_type": transaction["tx_type"]}
66          fortmatted_transaction["height"] =
                 transaction["height"]["value"]
67          fortmatted_transaction["total_input"] =
                 format_value(transaction["total_input"])
68          fortmatted_transaction["total_output"] =
                 format_value(transaction["total_output"])
69          fortmatted_transaction["inputs"] = [{i:inp[i] for i in inp if
                 i!='_request_tx_hash'} for inp in inputs if
                 inp["_request_tx_hash"] == transaction["tx_hash"]]
```

```
70          fortmatted_transaction["outputs"] = [{i:output[i] for i in
                 output if i!='_request_tx_hash'} for output in outputs if
                 output["_request_tx_hash"] == transaction["tx_hash"]]
71          formatted_transactions.append(fortmatted_transaction)
72      return formatted_transactions
73
74  def get_transactions_from_block_with_io(block_height):
75      transactions = blocks_api.list_block_txs('btc', block_height)
76      tx_hashes = [transaction["tx_hash"] for transaction in
             transactions]
77      inputs = get_io(tx_hashes, "inputs")
78      outputs = get_io(tx_hashes, "outputs")
79      return format_transactions(transactions, inputs, outputs)
80
81  def get_list_of_surrounding_blocks_for_number_of_transactions(
         block_height, number_of_transactions):
82      current_number_of_transactions = 0
83      current_block_height = block_height
84      blocks = []
85      increment = 1
86      while current_number_of_transactions < number_of_transactions:
87          blocks.append(current_block_height)
88          number_of_transactions_in_block = blocks_api.get_block('btc',
                 current_block_height)["no_txs"]
89          current_number_of_transactions +=
                 number_of_transactions_in_block
90          current_block_height = block_height + increment
91          increment = -increment if increment > 0 else -increment + 1
92      return sorted(blocks)
93
94  def find_block_by_timestamp_binary_search(min_block_height,
         max_block_height, timestamp, ensuing):
95      if max_block_height >= min_block_height:
96          mid_block_height = (max_block_height + min_block_height) // 2
97          mid_block = blocks_api.get_block('btc', mid_block_height)
98          if mid_block['timestamp'] == timestamp:
99              return mid_block_height
100         elif mid_block['timestamp'] > timestamp:
101             return
                     find_block_by_timestamp_binary_search(min_block_height,
                     mid_block_height - 1, timestamp, ensuing)
102         else:
103             return
                     find_block_by_timestamp_binary_search(mid_block_height
                     + 1, max_block_height, timestamp, ensuing)
104     else:
105         if ensuing:
106             return min_block_height
107         else:
108             return max_block_height
109
110 def create_data_set_1():
```

```
111     data_set_1 = db['data-set-1']
112     graphsense_statistics = general_api.get_statistics()
113     latest_block_height = [statistic['no_blocks'] - 1 for statistic in
             graphsense_statistics['currencies'] if statistic['name'] ==
             'btc'][0]
114     for block_height in range(100000, latest_block_height, 100000):
115         block_list =
                 get_list_of_surrounding_blocks_for_number_of_transactions(
                 block_height, 10000)
116         for block_height in tqdm(block_list):
117             if data_set_1.count_documents({"height": block_height}) ==
                     0:
118                 transactions =
                         get_transactions_from_block_with_io(block_height)
119                 data_set_1.insert_many(transactions)
120 
121 def create_data_set_2():
122     data_set_2 = db['data-set-2']
123     graphsense_statistics = general_api.get_statistics()
124     latest_block_height = [statistic['no_blocks'] - 1
125                            for statistic in
                                   graphsense_statistics['currencies'] if
                                   statistic['name'] == 'btc'][0]
126     print(latest_block_height)
127     start_date = datetime.datetime(2022, 2, 1).timestamp()
128     end_date = datetime.datetime(2022, 2, 3).timestamp()
129     start_block = find_block_by_timestamp_binary_search(
130         0, latest_block_height, start_date, True)
131     end_block = find_block_by_timestamp_binary_search(
132         start_block, latest_block_height, end_date, False)
133     block_list = list(range(start_block, end_block+1))
134     for block_height in tqdm(block_list):
135             if data_set_2.count_documents({"height": block_height}) ==
                     0:
136                 transactions =
                         get_transactions_from_block_with_io(block_height)
137                 data_set_2.insert_many(transactions)
```

C.2 Common-input-ownership heuristics

```
import pymongo
import os
from dotenv import load_dotenv
from tqdm import tqdm

load_dotenv('.env')

connectURI = os.environ["connectURI"]
client = pymongo.MongoClient(connectURI)
db = client["master"]

def get_addresses(puts):
    addresses = []
    for put in puts:
        try:
            addresses.append(put["address"][0])
        except IndexError as e:
            continue
    return addresses

def has_self_change_address(inputs, outputs):
    input_addresses = get_addresses(inputs)
    output_addresses = get_addresses(outputs)
    for output_address in output_addresses:
        if output_address in input_addresses:
            return True
    return False

def common_input_address_clustering(transactions, entities_collection):
    entities_collection.create_index("address_cluster")
    for transaction in tqdm(transactions):
        input_addresses = get_addresses(transaction["inputs"])
        existing_entity = False
        for index, address in enumerate(input_addresses):
            #Try to find entity with address in cluster
            entity_with_shared_input_address =
                 entities_collection.find_one({"address_cluster":
                 address})
            if entity_with_shared_input_address:
                existing_entity = True
                entity_id = entity_with_shared_input_address['_id']
                entities_collection.update_one({ '_id': entity_id }, {
                    '$addToSet': { 'address_cluster': { '$each':
                         input_addresses}},
                    '$push': {'tx_hashes': transaction["tx_hash"]}})
                input_addresses = input_addresses[index+1:]
                # check for existing entitites with any of the input
                     addresses in cluster and combine them with this
                     one
                for input_address in input_addresses:
```

```
46                          exiting_entity_shared_address =
                                 entities_collection.find_one({"$and":
                                 [{"address_cluster": input_address}, {"_id":
                                 {"$ne":
                                 entity_with_shared_input_address['_id']}}]})
47                          if exiting_entity_shared_address:
48                              entities_collection.update_one({ '_id':
                                     entity_id }, {
49                                  '$addToSet': { 'address_cluster': {
                                         '$each':
                                         exiting_entity_shared_address[
                                         "address_cluster"]}},
50                                  '$push': {'tx_hashes': { '$each':
                                         exiting_entity_shared_address[
                                         "tx_hashes"]}}})
51                              entities_collection.delete_one({"_id":
                                     exiting_entity_shared_address["_id"]})
52                      break
53          if not existing_entity and len(input_addresses) > 0:
54              entities_collection.insert_one({"address_cluster":
                     list(dict.fromkeys(input_addresses)), "tx_hashes":
                     [transaction["tx_hash"]]})
55
56  def individual_input_address_clustering(transactions,
         entities_collection):
57      entities_collection.create_index("address_cluster")
58      for transaction in tqdm(transactions):
59          input_addresses = get_addresses(transaction["inputs"])
60          for input_address in input_addresses:
61              if not entities_collection.find_one({"address_cluster":
                     input_address}):
62                  entities_collection.insert_one({"address_cluster":
                         [input_address], "tx_hashes":
                         [transaction["tx_hash"]]})
63
64
65  def cursor_to_list(cursor):
66      return [item for item in cursor]
67
68  def h1_1(transactions_collection, entities_collection):
69      transactions_common_input =
             cursor_to_list(transactions_collection.find({"coinbase":
             False}))
70      common_input_address_clustering(transactions_common_input,
             entities_collection)
71
72  def h1_2(transactions_collection, entities_collection):
73      #Find all transactions with one output, and cluster the inputs
74      transactions_common_input =
             cursor_to_list(transactions_collection.find({ "outputs": {
             "$size": 1}}))
```

```
75      common_input_address_clustering(transactions_common_input,
            entities_collection)
76
77      #Find transactions with more than one output and crate individual
            clusters for each address
78      transactions_individual_input =
            cursor_to_list(transactions_collection.find({ "$expr" : {
            "$gt" : [{ "$size" : "$outputs" } , 1]}}))
79      individual_input_address_clustering(transactions_individual_input,
            entities_collection)
80
81
82  def h1_3(transactions_collection, entities_collection, otc_collection):
83      #Find all transactions with one output
84      transactions_common_input =
            cursor_to_list(transactions_collection.find({ "outputs": {
            "$size": 1}}))
85
86      #Find all transactions with more than two outputs
87      transactions_individual_input =
            cursor_to_list(transactions_collection.find({ "$expr" : {
            "$gt" : [{ "$size" : "$outputs" } , 2]}}))
88
89      #Find transactions with two outputs and check for change address
            from otc_collection
90      transactions_two_outputs =
            cursor_to_list(transactions_collection.find({ "outputs": {
            "$size": 2}}))
91      for transaction in transactions_two_outputs:
92          if has_self_change_address(transaction) or
                otc_collection.find_one({"$and": [{"tx_hash":
                transaction["tx_hash"]}, {"heuristics.4": True}]}):
93              transactions_common_input.append(transaction)
94          else:
95              transactions_individual_input.append(transaction)
96
97      common_input_address_clustering(transactions_common_input,
            entities_collection)
98      individual_input_address_clustering(transactions_individual_input,
            entities_collection)
99
100
101
102 transactions_collection = db['data-set-2']
103 entities_collection = db['entities-data-set-2']
104 otc_collection = db['otc-addresses-data-set-2']
105
106 h1_3(transactions_collection, entities_collection, otc_collection)
```

C.3 One-time change address heuristics

```
import os
from time import sleep
import graphsense
import pymongo
from dotenv import load_dotenv
from graphsense.api import addresses_api, bulk_api, txs_api,
    entities_api
from tqdm import tqdm

load_dotenv('.env')

api_key = os.environ.get("api_key")
configuration =
    graphsense.Configuration(host="https://api.graphsense.info")
configuration.api_key["api_key"] = api_key

api_client = graphsense.ApiClient(configuration)

addresses_api = addresses_api.AddressesApi(api_client)
bulk_api = bulk_api.BulkApi(api_client)
txs_api = txs_api.TxsApi(api_client)
entities_api = entities_api.EntitiesApi(api_client)

connectURI = os.environ["connectURI"]
client = pymongo.MongoClient(connectURI)
db = client["master"]

def is_coin_generation(transaction):
    return transaction['coinbase']

def has_two_outputs(transaction):
    return len(transaction["outputs"]) == 2

def has_two_inputs(transaction):
    return len(transaction["inputs"]) == 2

def get_addresses(puts, json=False):
    addresses = []
    for put in puts:
        try:
            if json:
                addresses.append(put["address"][0][""])
            else:
                addresses.append(put["address"][0])
        except IndexError as e:
            continue
    return addresses

def has_self_change_address(transaction):
    input_addresses = get_addresses(transaction["inputs"])
    output_addresses = get_addresses(transaction["outputs"])
```

```
50      for output_address in output_addresses:
51          if output_address in input_addresses:
52              return True
53      return False
54
55  def get_first_transaction_hash(address):
56      return addresses_api.get_address('btc',
             address)["first_tx"]["tx_hash"]
57
58  def get_latest_output_transaction_binary_search(transactions, low,
         high):
59      if high >= low:
60          mid = (high + low) // 2
61          if transactions[mid].value.value > 0 and
                 transactions[mid-1].value.value < 0:
62              return transactions[mid]["tx_hash"]
63          elif transactions[mid].value.value > 0:
64              return get_latest_output_transaction_binary_search(
                     transactions, low, mid - 1)
65          else:
66              return get_latest_output_transaction_binary_search(
                     transactions, mid+1, high)
67      else:
68          return False
69
70  def is_used_as_output_later(address, current_transaction_tx_hash):
71      response = addresses_api.list_address_txs(
72          'btc', address, pagesize=500)
73      if response["address_txs"][0]["tx_hash"] ==
             current_transaction_tx_hash:
74          return False
75      while response:
76          transactions = response["address_txs"]
77          if transactions[-1].value.value > 0:
78              return get_latest_output_transaction_binary_search(
                     transactions, 0, len(transactions)-1) !=
                     current_transaction_tx_hash
79          response = addresses_api.list_address_txs('btc', address,
                 page=response['next_page'], pagesize=500)
80
81  def value_has_more_than_four_digits_after_dot(value):
82      value = '{0:.8f}'.format(value * 10**(-8)).strip("0")
83      number_of_decimals = len(value.split(".")[1])
84      return number_of_decimals > 4
85
86  def otc_value_is_smaller_than_all_input_values(otc_value, inputs):
87      input_values = [inp["value"]["value"] for inp in inputs]
88      for input_value in input_values:
89          if input_value < otc_value:
90              return False
91      return True
92
```

```
93  def has_not_been_otc_addressed_previously_h2_3(address):
94      first_transaction_for_address_hash =
            get_first_transaction_hash(address)
95      transaction_data = txs_api.get_tx('btc',
            first_transaction_for_address_hash, include_io=True)
96      transaction = {"tx_hash": transaction_data["tx_hash"], "coinbase":
            transaction_data["coinbase"], "outputs":
            transaction_data["outputs"].value, "inputs":
            transaction_data["inputs"].value}
97
98      if transaction['coinbase']:
99          return True
100
101     if not has_two_outputs(transaction):
102         return True
103
104     if has_two_inputs(transaction):
105         return True
106
107     otc_output = [output for output in transaction["outputs"] if
            output["address"][0] == address][0]
108     other_output =  [output for output in transaction["outputs"] if
            output["address"][0] != address][0]
109
110     if not value_has_more_than_four_digits_after_dot(
            otc_output["value"]["value"]):
111         return True
112
113     if has_self_change_address(transaction):
114         return True
115
116     first_transaction_for_other_address_hash =
            get_first_transaction_hash(other_output["address"][0])
117     return transaction["tx_hash"] ==
            first_transaction_for_other_address_hash
118
119 def has_not_been_otc_addressed_previously_h2_4(address):
120     first_transaction_for_address_hash =
            get_first_transaction_hash(address)
121     transaction_data = txs_api.get_tx('btc',
            first_transaction_for_address_hash, include_io=True)
122     transaction = {"tx_hash": transaction_data["tx_hash"], "coinbase":
            transaction_data["coinbase"], "outputs":
            transaction_data["outputs"].value, "inputs":
            transaction_data["inputs"].value}
123
124     if is_coin_generation(transaction):
125         return True
126
127     if has_self_change_address(transaction):
128         return True
129
```

```
130     if not has_two_outputs(transaction):
131         return True
132
133     if has_two_inputs(transaction):
134         return True
135
136     otc_output = next(output for output in transaction["outputs"] if
            output["address"][0] == address)
137     if not otc_value_is_smaller_than_all_input_values(
            otc_output["value"]["value"], transaction["inputs"]):
138         return True
139
140     other_output =  next(output for output in transaction["outputs"]
            if output["address"][0] != address)
141     first_transaction_for_other_address_hash =
            get_first_transaction_hash(other_output["address"][0])
142
143     if transaction["tx_hash"] !=
            first_transaction_for_other_address_hash:
144         return False
145     return not is_used_as_output_later(other_output["address"][0],
            transaction['tx_hash']) or is_used_as_output_later(address,
            transaction['tx_hash'])
146
147 def h2_123(transaction):
148     otc_data = {
149         "tx_hash": transaction["tx_hash"],
150         "block_height": transaction["height"],
151         "otc_output": None,
152         "other_output": None,
153         "heuristics": {
154             "1": False,
155             "2": False,
156             "3": False
157         }
158     }
159
160     # (2) The transaction T is not a coin generation;
161     if is_coin_generation(transaction):
162         return otc_data
163
164     # (4) The transaction T has exactly two outputs.
165     if not has_two_outputs(transaction):
166         return otc_data
167
168     # (3) There is no address among the outputs that also appears in
            the inputs (self-change address);
169     if has_self_change_address(transaction):
170         return otc_data
171
172     # (1) This is the first appearance of the OTC address;
173     output_1 = transaction["outputs"][0]
```

```
174     output_2 = transaction["outputs"][1]
175     output_address_1_first_transaction =
            get_first_transaction_hash(output_1["address"][0])
176     output_address_2_first_transaction =
            get_first_transaction_hash(output_2["address"][0])
177
178     is_first_transaction_of_output_address_1 =
            output_address_1_first_transaction == transaction['tx_hash']
179     is_first_transaction_of_output_address_2 =
            output_address_2_first_transaction == transaction['tx_hash']
180
181     if is_first_transaction_of_output_address_1 ==
            is_first_transaction_of_output_address_2 == False:
182         return otc_data
183
184     # (H2.2 (5)) Only the other output address is reused as an output
            address in some later transaction
185     elif is_first_transaction_of_output_address_1 ==
            is_first_transaction_of_output_address_2 == True:
186         output_1_used_later =
                is_used_as_output_later(output_1["address"][0],
                transaction["tx_hash"])
187         output_2_used_later =
                is_used_as_output_later(output_2["address"][0],
                transaction["tx_hash"])
188         if output_1_used_later != output_2_used_later:
189             if output_1_used_later:
190                 otc_data["otc_output"] = output_2
191                 otc_data["other_output"] = output_1
192                 otc_data["heuristics"]["2"] = True
193             elif output_2_used_later:
194                 otc_data["otc_output"] = output_1
195                 otc_data["other_output"] = output_2
196                 otc_data["heuristics"]["2"] = True
197
198     # (H2.1 (5) and H2.3 (5)) This is not the first appearance of the
            other output address
199     else:
200         if is_first_transaction_of_output_address_1:
201             otc_data["otc_output"] = output_1
202             otc_data["other_output"] = output_2
203             otc_data["heuristics"]["1"] = True
204         elif is_first_transaction_of_output_address_2:
205             otc_data["otc_output"] = output_2
206             otc_data["other_output"] = output_1
207             otc_data["heuristics"]["1"] = True
208
209         if is_used_as_output_later(
                otc_data["other_output"]["address"][0],
                transaction["tx_hash"]) and not is_used_as_output_later(
                otc_data["otc_output"]["address"][0],
                transaction["tx_hash"]):
```

```
210                 otc_data["heuristics"]["2"] = True
211 
212             # (H2.3 (6)) The number of inputs in transaction T is not
                     equal to two.
213             if has_two_inputs(transaction):
214                 return otc_data
215 
216             # (H2.3 (8)) Decimal representation of the value for the OTC
                     address has more than 4 digits after the dot.
217             if not value_has_more_than_four_digits_after_dot(
                     otc_data["otc_output"]["value"]["value"]):
218                 return otc_data
219 
220             # (H2.3 (7)) ~O has not been OTC addressed in previous
                     transactions
221             if has_not_been_otc_addressed_previously_h2_3(
                     otc_data["other_output"]["address"][0]):
222                 otc_data["heuristics"]["3"] = True
223         return otc_data
224 
225     def h2_4(transaction):
226         otc_data = {
227             "tx_hash": transaction["tx_hash"],
228             "block_height": transaction["height"],
229             "otc_output": None,
230             "other_output": None,
231             "heuristics": {
232                 "4": False
233             }
234         }
235 
236         # (2) The transaction T is not a coin generation;
237         if is_coin_generation(transaction):
238             return otc_data
239 
240         # (4) The transaction T has exactly two outputs.
241         if not has_two_outputs(transaction):
242             return otc_data
243 
244         # (3) There is no address among the outputs that also appears in
                 the inputs (self-change address);
245         if has_self_change_address(transaction):
246             return otc_data
247 
248         output_1 = transaction["outputs"][0]
249         output_2 = transaction["outputs"][1]
250 
251         is_first_transaction_of_output_address_1 =
                 get_first_transaction_hash(output_1["address"][0]) ==
                 transaction['tx_hash']
252         is_first_transaction_of_output_address_2 =
                 get_first_transaction_hash(output_2["address"][0]) ==
```

```
        transaction['tx_hash']
253
254     # (1) This is the first appearance of the OTC address;
255     if is_first_transaction_of_output_address_1 ==
            is_first_transaction_of_output_address_2 == False:
256         return otc_data
257
258     # (5) This is not the first appearance of the other output address
            OR only the other output address is reused as an output
            address in some later transaction
259     elif is_first_transaction_of_output_address_1 ==
            is_first_transaction_of_output_address_2 == True:
260         output_1_used_later =
                is_used_as_output_later(output_1["address"][0],
                transaction["tx_hash"])
261         output_2_used_later =
                is_used_as_output_later(output_2["address"][0],
                transaction["tx_hash"])
262         if output_1_used_later == output_2_used_later:
263             return otc_data
264         else:
265             if output_1_used_later:
266                 otc_output = output_2
267                 other_output = output_1
268             elif output_2_used_later:
269                 otc_output = output_1
270                 other_output = output_2
271     else:
272         if is_first_transaction_of_output_address_1:
273             otc_output = output_1
274             other_output = output_2
275         elif is_first_transaction_of_output_address_2:
276             otc_output = output_2
277             other_output = output_1
278
279         # (6) The other output address has not been OTC addressed in
                previous transactions
280         if not has_not_been_otc_addressed_previously_h2_4(
                other_output["address"][0]):
281             return otc_data
282
283     # (7) The one-time change value is smaller than any of the inputs
284     if not otc_value_is_smaller_than_all_input_values(
            otc_output["value"]["value"], transaction["inputs"]):
285         return otc_data
286
287     otc_data["otc_output"] = otc_output
288     otc_data["other_output"] = other_output
289     otc_data["heuristics"]["4"] = True
290
291     return otc_data
292
```

```
293
294 def run_otc_heuristic(heuristic_function, transactions_collection,
     otc_collection):
295     transactions = transactions_collection.find({})
296     transactions = [transaction for transaction in transactions]
297     for transaction in tqdm(transactions):
298         if otc_collection.count_documents({"tx_hash":
              transaction["tx_hash"]}) == 0:
299             try:
300                 otc_data = heuristic_function(transaction)
301             except graphsense.ApiException as e:
302                 print("Exception␣when␣calling␣
                     AddressesApi->list_address_txs:",
303                     e.status, e.reason)
304                 continue
305             except IndexError as e:
306                 print("\nException,␣probably␣empty␣address␣array.␣
                     tx_hash:", transaction["tx_hash"])
307                 continue
308             otc_collection.insert_one(otc_data)
309
310 if __name__ == '__main__':
311     run_otc_heuristic(h2_123, db['data-set-1'],
          db['otc-addresses-data-set-1'])
312     run_otc_heuristic(h2_4, db['data-set-2'],
          db['otc-addresses-data-set-2'])
```

C.4 Combination of the common-input-ownership heuristic and OTC heuristic

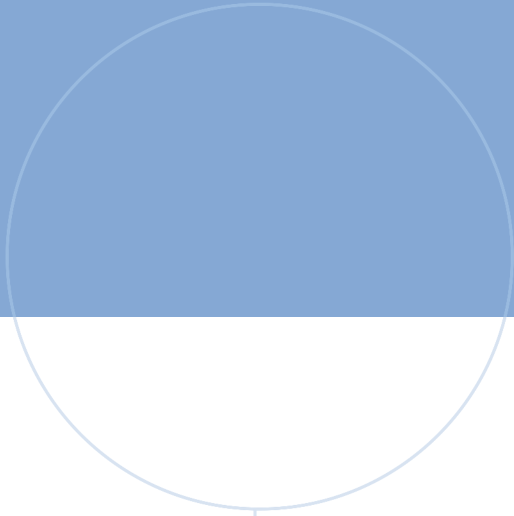
```
import pymongo
import os
from dotenv import load_dotenv
from tqdm import tqdm

load_dotenv('.env')

connectURI = os.environ["connectURI"]
client = pymongo.MongoClient(connectURI)
db = client["master"]

def combine_entities_with_otc_address(entities_collection,
     otc_collection):
    entities = entities_collection.find({})
    entities = [entity for entity in entities]
    for entity in tqdm(entities):
        for tx_hash in entity["tx_hashes"]:
            otc = otc_collection.find_one({"$and": [{"tx_hash":
                 tx_hash}, {"heuristics.4": True}]})
            if otc and otc["otc_output"]["address"][0] not in
                 entity["address_cluster"]:
                entity_with_otc_address =
                     entities_collection.find_one({"address_cluster":
                     otc["otc_output"]["address"][0]})
                if entity_with_otc_address and
                     entity_with_otc_address["_id"] != entity["_id"]:
                    entities_collection.update_one({
                        '_id': entity['_id']
                    }, {
                        '$addToSet': {
                            'address_cluster': { '$each':
                                 entity_with_otc_address[
                                 "address_cluster"]}
                        },
                        '$push': {
                            'tx_hashes': { '$each':
                                 entity_with_otc_address["tx_hashes"]}
                        }
                    })
                    entities_collection.delete_one({"_id":
                         entity_with_otc_address["_id"]})

entities_collection = db['entities-data-set-2']
otc_collection = db['otc-addresses-data-set-2']

combine_entities_with_otc_address(entities_collection, otc_collection)
```

C.5 Testing

```
1  import unittest
2  from heuristics_otc import value_has_more_than_four_digits_after_dot ,
       get_addresses , has_self_change_address ,
       otc_value_is_smaller_than_all_input_values
3
4  class Test(unittest.TestCase):
5      def test_value_has_more_than_four_decimals_after_dot(self):
6          self.assertFalse(value_has_more_than_four_digits_after_dot(
               123450000))
7          self.assertFalse(value_has_more_than_four_digits_after_dot(
               100010000))
8          self.assertTrue(value_has_more_than_four_digits_after_dot(
               123456000))
9          self.assertFalse(value_has_more_than_four_digits_after_dot(
               100000000))
10         self.assertTrue(value_has_more_than_four_digits_after_dot(1000))
11         self.assertFalse(value_has_more_than_four_digits_after_dot(
               10000))
12         self.assertTrue(value_has_more_than_four_digits_after_dot(1))
13         self.assertFalse(value_has_more_than_four_digits_after_dot(0))
14
15     def test_get_addresses(self):
16         puts = [ { "address" : [ "address1" ] } ]
17         self.assertEqual(get_addresses(puts), ["address1"])
18
19         puts = [ { "address" : [ "address1" ] }, { "address" : [
               "address2" ] }  ]
20         self.assertEqual(get_addresses(puts), ["address1", "address2"])
21
22         puts = [ { "address" : [ "address1" ] }, { "address" : [] }  ]
23         self.assertEqual(get_addresses(puts), ["address1"])
24
25         puts = [ { "address" : [] }, { "address" : [] }  ]
26         self.assertEqual(get_addresses(puts), [])
27
28     def test_get_addresses_json(self):
29         puts = [ { "address" : [{"" : "address1" }] } ]
30         self.assertEqual(get_addresses(puts, json=True), ["address1"])
31
32         puts = [ { "address" : [{"" : "address1" }] }, { "address" :
               [{"" : "address2" }] }  ]
33         self.assertEqual(get_addresses(puts, json=True), ["address1",
               "address2"])
34
35         puts = [ { "address" : [{"" : "address1" }] }, { "address" :
               [] }  ]
36         self.assertEqual(get_addresses(puts, json=True), ["address1"])
37
38         puts = [ { "address" : [] }, { "address" : [] }  ]
39         self.assertEqual(get_addresses(puts, json=True), [])
40
```

```
41      def test_has_self_change_address(self):
42          transaction = {
43              "inputs": [ { "address" : [ "address1" ] } ],
44              "outputs": [ { "address" : [ "address1" ] } ]
45          }
46          self.assertTrue(has_self_change_address(transaction))
47
48          transaction = {
49              "inputs": [ { "address" : [ "address1" ] } ],
50              "outputs": [ { "address" : [ "address2" ] } ]
51          }
52          self.assertFalse(has_self_change_address(transaction))
53
54          transaction = {
55              "inputs": [ { "address" : [ "address1" ] }, { "address" :
                     [ "address2" ] } ],
56              "outputs": [ { "address" : [ "address3" ] }, { "address" :
                     [ "address1" ] } ]
57          }
58          self.assertTrue(has_self_change_address(transaction))
59
60          transaction = {
61              "inputs": [ { "address" : [ "address1" ] }, { "address" :
                     [ "address2" ] } ],
62              "outputs": [ { "address" : [ "address3" ] }, { "address" :
                     [ "address4" ] } ]
63          }
64          self.assertFalse(has_self_change_address(transaction))
65
66          transaction = {
67              "inputs": [ { "address" : [ "address1" ] } ],
68              "outputs": [ { "address" : [ "address1" ] }, { "address" :
                     [ "address2" ] } ]
69          }
70          self.assertTrue(has_self_change_address(transaction))
71
72          transaction = {
73              "inputs": [ { "address" : [ "address1" ] }, { "address" :
                     [ "address2" ] } ],
74              "outputs": [ { "address" : [ "address3" ] }, { "address" :
                     [ ] } ]
75          }
76          self.assertFalse(has_self_change_address(transaction))
77
78          transaction = {
79              "inputs": [ { "address" : [ ] } ],
80              "outputs": [ { "address" : [ "address1" ] }]
81          }
82          self.assertFalse(has_self_change_address(transaction))
83
84      def test_otc_value_is_smaller_than_all_input_values(self):
85          inputs = [{"value": {"value": 100000000}}]
```

```
86         self.assertTrue(otc_value_is_smaller_than_all_input_values(999,
               inputs))
87
88         inputs = [{"value": {"value": 999}}]
89         self.assertFalse(otc_value_is_smaller_than_all_input_values(
               1000, inputs))
90
91         inputs = [{"value": {"value": 1000}}, {"value": {"value":
               1001}}, ]
92         self.assertTrue(otc_value_is_smaller_than_all_input_values(999,
               inputs))
93
94         inputs = [{"value": {"value": 999}}, {"value": {"value":
               1001}}, ]
95         self.assertFalse(otc_value_is_smaller_than_all_input_values(
               1000, inputs))
96
97         inputs = [{"value": {"value": 1000}}, {"value": {"value":
               2000}}, {"value": {"value": 998}}, {"value": {"value":
               99912931299}},]
98         self.assertFalse(otc_value_is_smaller_than_all_input_values(
               999, inputs))
```



Håvard Hunshamar & Marcus Frenje

Techniques and tools for cryptocurrency intelligence and blockchain forensics

NTNU
Norwegian University of
Science and Technology